

原创经典，威盛一线工程师倾力打造

深入驱动核心，剖析操作系统底层运行机制

通过实例引导，快速学习编译、安装、调试的方法



Windows

驱动开发技术详解

Windows Driver Development Internals

张帆 史彩成 等编著



- ◎ 从Windows最基本的两类驱动程序的编译、安装、调试入手讲解，非常容易上手
- ◎ 用实例详细讲解PCI、USB、虚拟串口、虚拟摄像头、SDIO等驱动程序的开发
- ◎ 归纳了多种调试驱动程序的高级技巧，如用WinDbg和VMWare软件对驱动进行源码级调试
- ◎ 介绍了多种实用的工具软件，如BusHound、IRPTrace、DebugView等
- ◎ 深入Windows操作系统的底层和内核，透析Windows驱动开发的本质



CD-ROM



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

51 单片机

开发与应用技术详解

赵建领 薛园园 等编著

书名:	责编:
社名:	校次:
开本:	正文页码:
文前页码:	排版员:
日期:	排版公司:
主管签字:	

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书全面详细地讲述了单片机的原理、编程指南及应用案例，其中 51 系列单片机的编程部分是本书的重点。全书分为 5 篇 38 章。首先介绍了 51 系列单片机的开发概述及单片机的基本结构，接着介绍了单片机的汇编程序设计语言，然后介绍了单片机 C51 语言的程序设计，随后结合单片机的指令系统及各个功能部件详细讲解了单片机的编程操作，以及单片机硬件资源的仿真和程序调试。本书最后还给出了在各个领域中常用到的一些典型案例，供读者在学习和工作中参考。

本书知识点覆盖全面、结构安排紧凑、讲解详细、实例丰富。对于 51 系列单片机的初学者，通过本书可以快速掌握单片机的程序设计。本书对具有一定开发经验的设计人员，也有很好的参考价值。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

51 单片机开发与应用技术详解 / 赵建领等编著. —北京：电子工业出版社，2009.1
ISBN 978-7-121-07920-7

I. 5… II. 赵… III. 单片微型计算机 IV. TP368.1

中国版本图书馆 CIP 数据核字（2008）第 188134 号

责任编辑：高洪霞

印 刷：北京东光印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：44 字数：1235 千字

印 次：2009 年 1 月第 1 次印刷

印 数：4000 册 定价：79.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。



前言 Introduction

51 系列单片机是目前应用最为广泛的一类微处理器，它以低廉的价格和强大的功能，受到广大电子设计爱好者和工程师的欢迎。51 系列单片机内部具有丰富的硬件资源，例如定时器/计数器、中断系统、串行接口，并且它还提供了详尽的指令操作系统，可以供程序员很方便地进行程序设计。在 51 系列单片机的开发过程中，程序设计是重点也是难点。初学者往往很难快速掌握单片机指令的应用、各个功能部件的编程方法及程序设计思路。本书重点针对 51 系列单片机的编程进行阐述，详细讲解各个指令及功能部件的编程方法，并给出大量的程序示例供读者学习参考。除此之外，本书同样系统地介绍了 51 系列单片机的结构和原理，最后还给出了一些典型的应用案例。

和其他书籍相比，本书有如下优点

- ① 本书首先详细介绍了 51 系列单片机的基础知识，然后对单片机的编程语言及程序设计方式进行了详细的讲解，接着还介绍了很多具有代表性的案例。
- ② 本书在讲解每个知识点的同时，均给出了其在程序设计中的应用实例，每个实例都可以仿真执行，读者可以快速掌握对应知识点在程序设计中的应用。
- ③ 本书不局限于一种编程语言，对汇编语言和流行的单片机 C 语言都做了详细介绍，其中以应用最为广泛的单片机 C 语言作为重点，而且在绝大部分的知识点讲解中给出了汇编语言和单片机 C 语言的程序示例，部分知识点则根据程序本身的需要，选择一种更为方便的实现方式。
- ④ 本书不仅介绍了基本的程序设计方式，还介绍了应用于 51 系列单片机的 RTX-51 实时多任务操作系统。
- ⑤ 本书案例丰富，基本上涵盖了电子设计的各个领域，如键盘接口、LED、LCD 液晶显示、CPLD、I2C 总线、实时时钟、音乐播放、RAM 存储器读写、RTX-51 实时多任务操作系统、温度传感器、打印机、A/D 转换和 D/A 转换等。
- ⑥ 本书对每一个案例都详细介绍了知识背景、硬件及软件设计流程，并对程序代码进行了详细的注释，对开始的案例几乎是逐行注释，使之更加容易理解。

本书的内容

本书把实用性、系统性和完整性作为重点，详细介绍了 51 系列单片机的原理和功能，对于每一个知识点均给出了详细的程序设计方法和编程示例，最后还提供了一些典型的应用案例。本书内容分为 5 篇，共 38 章。

第一篇是 51 系列单片机基础，共分为三章，详细介绍了 51 系列单片机的发展、开发流程、基本结构和 Keil C51 开发工具等。

第一篇	第 1 章 51 系列单片机概述	→	主要介绍了 51 系列单片机的发展、最新 51 内核单片机和 51 系列单片机的开发概述。
	第 2 章 51 系列单片机基本结构	→	主要介绍了 51 系列单片机基本结构、引脚功能、存储器结构、复位及时序 以及掉电保护和低功耗设计等。
	第 3 章 Keil C51 开发工具简介	→	主要介绍了 Keil μ Vision3 的安装及集成开发环境。

第二篇是编程篇——汇编语言，共分为两章，详细介绍了单片机汇编语言的程序设计和程序结构。

第二篇	第 4 章 汇编语言程序设计	→	主要介绍了汇编语言指令和格式，并给出了 Keil μ Vision3 集成开发环境中运行汇编语言程序的实例。
	第 5 章 汇编语言程序结构	→	主要介绍了汇编语言的各种程序结构。

第三篇是编程篇——C51，共分为 10 章，重点介绍了单片机 C51 语言的程序设计。

第三篇	第 6 章 单片机 C 语言程序设计基础	→	主要介绍了 C51 语言的标识符、数据类型、变量作用域、运算符、表达式等。
	第 7 章 数组	→	主要介绍了 C51 中的数组，包括数组的声明表示，一维数组、二维数组和多维数组的使用。
	第 8 章 指针	→	主要介绍了 C51 中的指针，包括指针的声明使用及指针变量，并介绍了其他一些特殊用途的指针。
	第 9 章 结构	→	主要介绍了 C51 中的结构，包括结构及结构变量的使用。
	第 10 章 联合、枚举、类型说明和位域	→	主要介绍了 C51 语言中的联合、枚举、类型说明和位域等特殊数据类型的使用。
	第 11 章 C51 语言的函数	→	主要介绍了函数的定义、调用和 main 函数。
	第 12 章 C51 语言的常用库函数详解	→	主要对 Keil μ Vision3 集成开发环境下常用的库函数进行了详细的介绍。
	第 13 章 C51 语句和流程	→	主要介绍了 C51 语言中的说明语句、表达式语句、复合语句、条件语句、开关判断语句，以及循环语句和函数调用语句，并介绍了用 C51 语言进行程序设计的常用流程控制结构。
	第 14 章 预处理及用户配置文件	→	主要介绍了 C51 中的各种预处理指令和控制参数，并介绍了 C51 的常用配置文件，包括启动代码文件、基本 I/O 函数文件等。
	第 15 章 C51 语言的存储结构	→	主要介绍了 51 系列单片机的存储单元，C51 中的存储类型、基于存储器的指针、动态存储分配等。

第四篇是 51 系列单片机编程指南篇，共分为 7 章，详细讲解了单片机的指令系统、定时器/计数器、串行接口、中断及 RTX-51 实时多任务操作系统。在讲解的过程中，对每一个知识点都提供了详细的程序设计方法和程序示例。最后还对 Keil μ Vision3 集成开发环境下的单片机仿真和调试进行了介绍。

第四篇	第 16 章 51 系列单片机的指令系统	主要介绍了指令的各种寻址方式，以及数据传送指令、算术运算指令、逻辑运算及移位指令、控制转移指令和位操作指令等，并给出了具体的程序示例。
	第 17 章 51 系列单片机的定时器/计数器	主要介绍了定时器/计数器的内部结构、控制寄存器，以及各种工作模式及其程序设计方法等。
	第 18 章 51 系列单片机中断系统及其程序设计	主要介绍了中断的类型、中断系统的控制寄存器及其程序访问、中断的处理过程，以及外部中断源的扩展等。
	第 19 章 51 系列单片机的串行接口	主要介绍了串行接口的通信方式、单片机的串行口结构及程序控制，并详细介绍了串行口的 4 种工作模式及其程序示例。
	第 20 章 C51 下的 RTX-51 实时多任务操作系统	主要介绍了 RTX-51 的任务调度、系统函数及任务管理等。
	第 21 章 Keil μ Vision3 中的单片机硬件资源仿真	主要介绍了如何在 Keil μ Vision3 集成开发环境下，仿真单片机各种片上资源。
	第 22 章 Keil μ Vision3 中的程序调试	主要介绍了如何在 Keil μ Vision3 集成开发环境下进行程序调试及常用的命令调试。

第五篇是典型案例篇，共分为 16 章，详细介绍了单片机在一些常用领域中的应用，对每一个应用均给出了完整的电路图和程序分析。

第五篇	第 23 章 键盘程序设计	主要介绍了常用的键盘结构、单片机与矩阵式键盘的编程接口，并给出了具体的实例。
	第 24 章 LED 数码管显示	主要介绍了 LED 数码管的种类和结构、数码管的静态和动态驱动方式，并给出了具体的设计实例。
	第 25 章 LCD 液晶显示模块	主要介绍了液晶的类型、液晶的驱动器，并通过一个常用的液晶显示器介绍了如何使用单片机来实现汉字和图形的显示。
	第 26 章 D/A 转换实例	主要介绍了 D/A 转换器的类型结构，并给出了一个性能优秀的 D/A 转换器及其程序操作实例。
	第 27 章 可编程逻辑器件 CPLD	主要介绍了可编程逻辑器件 CPLD 和 FPGA 的结构、VHDL 程序设计语言，并通过设计实例，介绍了如何使用 CPLD 来扩展单片机的接口。
	第 28 章 51 系列单片机读写 I2C 总线	主要介绍了 I2C 总线的工作原理、I2C 总线的传输协议及程序实现，最后通过具体的实例介绍了单片机读写 I2C 器件的操作。
	第 29 章 单片机音乐播放	主要介绍了单片机发音的原理，并给出了一个单片机实现音乐播放实例。

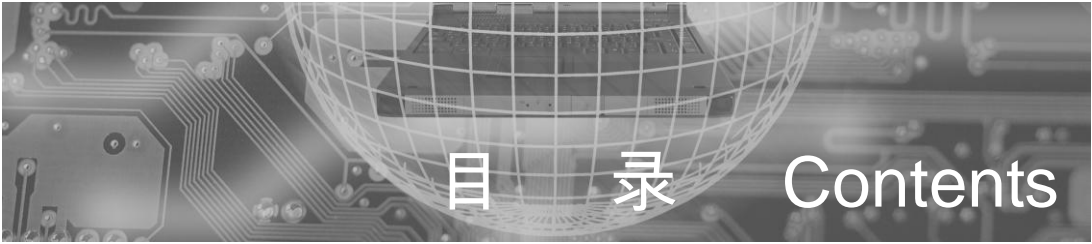
第 30 章 实时时钟芯片应用	主要介绍了常用的实时时钟芯片原理及数据传输方式,并给出了单片机实现时钟操作的实例。
第 31 章 静态 RAM 存储器应用	主要介绍了静态 RAM 存储器的读写,并通过一个实例实现了单片机读写外部 RAM 存储器,程序中还用到了单片机与计算机之间的通信通信。
第 32 章 道路交通灯控制系统	主要介绍了交通灯控制系统的原理,并通过 RTX-51 实时多任务操作系统来实现了道路交通的模拟控制。
第 33 章 单总线温度传感器 DS18S20	主要介绍了 1-Wire 单总线的传输协议及单总线接口温度传感器 DS18S20,最后给出了一个单片机模拟单总线读写 DS18S20 的实例。
第 34 章 Microware 串行总线 EEPROM 的应用	主要介绍了 Microware 串行总线,以及 Microware 串行总线接口的 EEPROM 存储器,最后给出了一个单片机模拟 Microware 串行总线来读写 EEPROM 的实例。
第 35 章 单片机控制打印机实例	主要介绍了打印机及其接口,并通过一个实例介绍了如何使用单片机控制打印机。
第 36 章 A/D 转换实例	主要介绍了 A/D 转换器的类型结构,并给出了一个 8 通道的 A/D 转换器及其程序操作实例。
第 37 章 单片机读写智能 IC 卡	主要介绍了智能 IC 卡,以及典型的智能 IC 卡芯片,最后通过一个实例介绍了如何使用单片机读写智能 IC 卡。
第 38 章 单片机智能锂电池充电管理	主要介绍了锂电池及其充电的特性,并介绍了一款高性能的锂电池充电管理芯片,最后通过一个实例介绍了如何使用单片机来实现锂电池智能充电。

本书结构紧凑,知识点涉及全面,内容翔实,案例丰富。由于本书内容较多,编写时间较仓促,所以书中难免有遗漏和不足之处,恳请广大读者提出宝贵意见,以便于笔者做进一步改进。

适合的读者

- 单片机的初学者
- 电子设计爱好者
- 电子工程师
- 系统开发人员
- 大学、大专等电子相关专业的学生及教师

编者



第一篇 51 系列单片机基础

第 1 章 51 系列单片机概述 35

本章主要介绍了单片机的发展历程、51 系列单片机的简介和 51 系列单片机的应用领域；接着介绍了一下实际常用的一些主流半导体厂商的单片机型号，以供读者参考；最后，详细介绍了开发 51 系列单片机应用系统的整个流程及主要注意事项。

- 1.1 单片机的产生与发展..... 35
- 1.2 51 系列单片机介绍..... 36
 - 1.2.1 51 系列单片机简介..... 36
 - 1.2.2 51 系列单片机的应用领域..... 36
- 1.3 最新 51 内核单片机介绍..... 37
 - 1.3.1 Atmel 单片机介绍..... 37
 - 1.3.2 Cypress 单片机介绍..... 37
 - 1.3.3 Infineon 单片机介绍..... 38
 - 1.3.4 Silicon 单片机介绍..... 38
 - 1.3.5 Maxim 单片机介绍..... 38
 - 1.3.6 NXP 单片机介绍..... 39
 - 1.3.7 Winbond 单片机介绍..... 39
 - 1.3.8 Analog Devices 单片机介绍..... 39
 - 1.3.9 TI 单片机介绍..... 40
 - 1.3.10 其他单片机介绍..... 40
- 1.4 51 系列单片机开发概述..... 40
 - 1.4.1 分析测控系统..... 41
 - 1.4.2 单片机选型..... 41
 - 1.4.3 硬件资源分配..... 41
 - 1.4.4 程序设计..... 42
 - 1.4.5 仿真测试..... 42
 - 1.4.6 实际硬件测试..... 42

1.5 小结	42
--------------	----

第 2 章 51 系列单片机基本结构

43

本章详细介绍了 51 系列单片机的基本结构，包括单片机的内部结构、引脚功能、中央处理器 CPU、存储器的结构、单片机的复位、单片机工作的时钟和时序、并行 I/O 端口的结构和性能，以及单片机系统的掉电保护和低功耗设计。最后，还给出了一个典型的单片机最小系统。本章在讲述的时候，不仅对基本的 8051 进行了介绍，还介绍了目前新推出的高性能单片机 AT89S52。这样读者在学习基础知识的同时，还可以了解最新单片机的性能和使用。

2.1 51 系列单片机的内部结构	43
2.1.1 内部结构的主要组成部分	43
2.1.2 单片机结构的类型	44
2.2 51 系列单片机的引脚功能	45
2.2.1 51 系列单片机的引脚封装	45
2.2.2 51 系列单片机引脚功能	45
2.3 中央处理器（CPU）	47
2.3.1 算术逻辑部件（ALU）	47
2.3.2 控制器	47
2.3.3 通用寄存器	48
2.3.4 专用寄存器	48
2.4 存储器结构	50
2.4.1 程序存储器及其扩展	50
2.4.2 数据存储器及其扩展	51
2.5 51 系列单片机的复位	55
2.5.1 单片机的复位状态	55
2.5.2 单片机的复位电路	55
2.6 51 系列单片机的时钟和时序	56
2.6.1 振荡器和时钟电路	57
2.6.2 CPU 的时序	58
2.6.3 指令执行的时序	58
2.6.4 访问外部 ROM/RAM 的操作时序	60
2.7 51 系列单片机的并行 I/O 口	61
2.7.1 并行 I/O 口	61
2.7.2 并行 I/O 口的应用	64
2.7.3 并行 I/O 口的扩展	64
2.8 系统掉电保护和低功耗设计	66
2.8.1 掉电保护	66
2.8.2 低功耗设计	67
2.9 51 系列单片机的最小系统	68

2.10 小结	69
---------	----

第 3 章 Keil C51 开发工具简介

70

本章详细介绍了 Keil μ Vision3 的集成开发环境 μ Vision3，包括安装过程、项目管理窗口、菜单栏、工具栏，以及 μ Vision3 的管理配置和各种常用的窗口。 μ Vision3 是一个十分优秀的单片机开发软件，应用得十分广泛，熟练掌握 μ Vision3 集成开发环境的使用是单片机设计的基础。

3.1 Keil μ Vision3 简介	70
3.2 μ Vision3 安装	70
3.3 μ Vision3 集成开发环境	71
3.3.1 μ Vision3 项目管理窗口	72
3.3.2 μ Vision3 的菜单栏	72
3.3.3 μ Vision3 的工具栏	77
3.3.4 μ Vision3 的管理配置	79
3.3.5 μ Vision3 的各种常用窗口	81
3.4 小结	84

第二篇 编程篇——汇编语言

第 4 章 汇编语言程序设计

85

本章主要讲述了汇编语言的伪指令、汇编语言的程序格式及源程序的编译。本章最后还通过一个具体的实例，介绍了如何在 Keil μ Vision3 集成开发环境中进行单片机的汇编程序设计及仿真。汇编语言是最早应用于单片机开发的程序语言。相比其他程序设计语言，汇编语言指令的执行速度快、代码短小精悍，且具有确定的指令执行周期。熟练掌握本章内容是后续汇编语言程序学习的基础。

4.1 汇编语言程序概述	85
4.1.1 汇编语言简介	85
4.1.2 汇编语言程序设计步骤	86
4.1.3 汇编语言程序实例	86
4.2 伪指令	87
4.2.1 汇编程序起始伪指令 ORG	87
4.2.2 汇编程序结束伪指令 END	87
4.2.3 等值伪指令 EQU (或 =)	87
4.2.4 数据地址赋值伪指令 DATA	88
4.2.5 定义字节伪指令 DB	88
4.2.6 定义字伪指令 DW	88
4.2.7 定义空间伪指令 DS	89
4.2.8 位地址符号伪指令 BIT	89
4.3 汇编语言程序的格式	89

4.4 源程序的汇编..... 90

4.5 Keil μVision3 中运行汇编语言实例..... 90

4.5.1 创建项目 90

4.5.2 创建源文件 91

4.5.3 编译项目 92

4.5.4 仿真调试 92

4.6 小结 93

第 5 章 汇编语言程序结构

94

本章主要介绍了利用汇编语言进行单片机设计的各种程序结构，包括顺序结构、分支结构、循环结构和子程序结构。本章还介绍了两类比较常用的程序设计类型，即查表程序和运算类的程序。对于比较复杂的问题可以根据设计的要求，选用不同的程序结构来进行设计。因此，熟练掌握本章内容，对读者以后的设计工作会很有帮助。

5.1 顺序结构程序..... 94

5.2 分支结构程序..... 95

5.2.1 双分支结构 95

5.2.2 多分支结构 96

5.3 循环结构程序..... 97

5.3.1 循环程序的结构和组成..... 97

5.3.2 循环程序示例 98

5.3.3 多重循环程序结构..... 99

5.4 子程序结构 100

5.4.1 子程序的结构 100

5.4.2 子程序的调用与返回..... 100

5.4.3 子程序设计实例..... 101

5.5 查表结构程序..... 102

5.6 运算类程序 103

5.6.1 8 位带符号整数的乘法子程序..... 103

5.6.2 8 位带符号整数的除法子程序..... 104

5.7 小结 106

第三篇 编程篇——C51

第 6 章 单片机 C 语言程序设计基础

107

单片机 C 语言是现在单片机系统设计所广泛采用的程序语言。本章首先介绍了单片机 C 语言（C51 语言）的特点，然后分别介绍了 C51 语言的标识符、关键字、数据类型、运算符和表达式等，并对每一个知识点都提供了完整详细的实例分析。本章是单片机 C51 语言的基础，熟练掌握本章知识，对以后章

节的学习会有很大的帮助。

- 6.1 单片机 C 语言概述 107
 - 6.1.1 单片机 C 语言和汇编语言对比 107
 - 6.1.2 C51 语言的主要特点 107
- 6.2 单片机 C 语言在 Keil μ Vision3 中应用实例 108
 - 6.2.1 创建项目 108
 - 6.2.2 创建源文件 109
 - 6.2.3 编译项目 109
 - 6.2.4 仿真调试 110
- 6.3 C51 的标识符与关键字 111
 - 6.3.1 标识符 111
 - 6.3.2 关键字 111
- 6.4 C51 的数据类型 113
 - 6.4.1 常量与变量 114
 - 6.4.2 整型数据 114
 - 6.4.3 浮点型数据 115
 - 6.4.4 字符型数据 116
 - 6.4.5 指针型数据 119
 - 6.4.6 无值型数据 120
- 6.5 C51 的变量作用域 120
 - 6.5.1 基本规则 120
 - 6.5.2 自动变量 121
 - 6.5.3 外部变量 122
 - 6.5.4 静态变量 123
 - 6.5.5 寄存器变量 123
- 6.6 分隔符与 const 修饰符 123
 - 6.6.1 C51 分隔符 123
 - 6.6.2 const 修饰符 124
- 6.7 运算符 125
 - 6.7.1 算术运算符 125
 - 6.7.2 逻辑运算符和关系运算符 126
 - 6.7.3 位运算符 128
 - 6.7.4 特殊运算符 130
 - 6.7.5 运算符优先级和结合性 133
- 6.8 表达式 134
 - 6.8.1 算术表达式 134
 - 6.8.2 赋值表达式 135

6.8.3 逗号表达式	136
6.8.4 关系和逻辑表达式	137
6.9 小结	138

第 7 章 数组

139

本章首先介绍了 C51 语言中的数组的使用，主要包括数组类型的说明、数组元素的表示及初始化赋值。接着，本章分别对一维数组、二维数组及多维数组的使用进行了详细的讲解。数组是重要数据结构，广泛应用于 C51 语言的程序设计中。因此，熟练掌握本章内容是学习 C51 语言的基础。

7.1 数组类型说明	139
7.2 数组元素的表示	140
7.3 数组元素的初始化赋值	140
7.4 一维数组	141
7.4.1 一维数组声明	142
7.4.2 向函数传递一维数组	142
7.4.3 一维字符串数组	142
7.5 二维数组	143
7.5.1 二维数组声明	144
7.5.2 二维数组初始化	145
7.5.3 二维字符串数组	146
7.6 多维数组	147
7.7 小结	147

第 8 章 指针

148

本章首先介绍了 C51 语言中指针的使用，主要包括指针的概念、指针变量的声明和赋值。接着，本章详细讲解了指针变量的引用及运算规则。最后，本章还对几种特殊的指针进行了详细的介绍，主要包括数组指针、字符指针及指针数组。指针的概念来源于 C 语言，在程序设计中灵活使用指针可以优化程序设计。因此，熟练掌握本章内容是学习 C51 语言的基础，同时对读者以后的 C51 语言程序设计工作会大有帮助。

8.1 地址、指针和指针变量的概念	148
8.2 指针变量的声明	148
8.3 指针变量的赋值	149
8.3.1 初始化赋值	149
8.3.2 取地址赋值	149
8.3.3 指针之间赋值	149
8.3.4 数组赋值	150
8.3.5 字符串赋值	150
8.3.6 函数入口赋值	150
8.4 指针变量的引用	150

8.4.1	取地址运算符“&”	150
8.4.2	取内容运算符“*”	151
8.5	指针变量的运算	152
8.5.1	关系运算	152
8.5.2	算术运算	152
8.6	数组指针	153
8.6.1	指向一维数组的指针	153
8.6.2	指向二维数组的指针	154
8.6.3	指向一个由 n 个元素所组成的数组指针	156
8.6.4	指针和数组的关系	156
8.7	字符指针	157
8.8	指针数组	158
8.9	小结	159

第 9 章 结构

160

本章首先介绍了 C51 语言中的结构的使用，主要包括结构的定义、结构变量的定义和赋值。接着，本章详细讲解了结构数组及结构指针的使用。最后，本章还对几种特殊的指针进行了详细的介绍，主要包括嵌套结构和位结构。结构是 C51 语言中重要的数据类型，其和 C 语言中的结构非常类似。熟练掌握本章内容是学习 C51 语言的基础，同时对读者以后的 C51 程序设计工作会大有帮助。

9.1	结构的定义	160
9.2	结构变量的定义	161
9.2.1	先定义结构，再定义结构变量	161
9.2.2	在定义结构的同时，定义结构变量	161
9.2.3	直接说明结构变量	161
9.3	结构变量的使用	162
9.4	多重结构变量的赋值	163
9.5	结构变量的初始化	163
9.6	结构数组	164
9.6.1	结构数组的定义	164
9.6.2	结构数组的初始化赋值	165
9.7	结构指针	166
9.8	特殊结构	168
9.8.1	嵌套结构	168
9.8.2	位结构	169
9.9	小结	170

第 10 章 联合、枚举、类型说明和位域

171

本章主要介绍了几种特殊形式的聚合数据类型，包括联合类型、枚举类型，还介绍了类型说明及位

域。这些特殊的数据类型是对基本数据类型的有效扩充。灵活运用这些特殊数据类型，可以方便单片机的程序设计。因此，熟练掌握本章内容是单片机程序设计的基础。

10.1 联合类型	171
10.1.1 联合和联合变量的定义	171
10.1.2 联合变量成员的引用	172
10.2 结构和联合的区别	174
10.3 枚举类型	175
10.3.1 枚举的定义	175
10.3.2 枚举变量的声明	175
10.3.3 枚举类型变量的赋值	176
10.4 类型说明	178
10.5 位域	179
10.5.1 位域的定义和位域变量的声明	179
10.5.2 位域变量的使用	180
10.6 小结	181

第 11 章 C51 语言的函数

182

本章详细介绍了 C51 语言中函数的使用，包括函数的定义、形参和实参、函数的返回值、函数的调用、函数及其变量的作用域。本章还结合单片机程序设计的特点，介绍了 C51 程序中的 main 函数。函数是 C51 语言中的重要概念，灵活运用函数可以实现程序的模块化设计。因此，熟练掌握本章内容，是 C51 语言程序设计的基础。

11.1 函数的概念和分类	182
11.1.1 从函数定义角度	182
11.1.2 从有无返回值角度	183
11.1.3 从数据传送角度	183
11.2 函数的定义	183
11.3 函数的参数	185
11.3.1 形参和实参	185
11.3.2 数组作为函数参数	186
11.3.3 多维数组作为函数参数	188
11.3.4 指针作为函数参数	188
11.4 函数的返回值	189
11.5 函数调用	189
11.5.1 赋值调用与引用调用	190
11.5.2 递归调用	190
11.5.3 嵌套调用	192
11.6 函数及其变量的作用域	193
11.6.1 函数的作用域	193

11.6.2 函数的变量作用域	193
11.7 main 函数	194
11.8 小结	194

第 12 章 C51 语言的常用库函数详解 195

本章详细讲解了 C51 语言中常用的库函数，主要包括字符函数、字符串函数、I/O 函数、数学函数、标准函数、内部函数、绝对地址访问函数、变量参数表函数、全程跳转函数及偏移量函数。这些库函数涵盖了常见的字符、字符串、数学计算、I/O 控制等功能，读者在进行程序设计时可以直接调用使用。因此，熟练掌握和运用本章内容，可以大大减轻程序设计的负担，方便单片机的程序设计。

12.1 字符函数	195
12.1.1 检查英文字母函数	195
12.1.2 检查字母数字函数	196
12.1.3 检查控制字符函数	196
12.1.4 十进制数字检查函数	197
12.1.5 可打印字符检查函数	197
12.1.6 包含空格的可打印字符检查函数	198
12.1.7 格式字符检查函数	199
12.1.8 小写英文字母检查函数	199
12.1.9 大写英文字母检查函数	200
12.1.10 控制字符检查函数	200
12.1.11 十六进制数字检查函数	201
12.1.12 十六进制数字转换函数	202
12.1.13 大写字符转换函数	202
12.1.14 小写字符转换函数	203
12.1.15 ASCII 字符转换函数	204
12.1.16 大写字符宏转换函数	204
12.1.17 小写字符宏转换函数	205
12.2 字符串函数	206
12.2.1 字符查找函数	206
12.2.2 指定长度的字符串比较函数	206
12.2.3 字符串复制函数	207
12.2.4 带终止字符的字符串复制函数	208
12.2.5 字符串移动函数	209
12.2.6 字符串填充函数	209
12.2.7 字符串追加函数	210
12.2.8 指定长度的字符串追加函数	210
12.2.9 字符串比较函数	211

12.2.10	包含结束符的字符串比较函数	212
12.2.11	字符串覆盖函数	213
12.2.12	指定长度的字符串覆盖函数	213
12.2.13	获取字符串个数函数	214
12.2.14	搜索字符串函数	214
12.2.15	搜索字符函数	215
12.2.16	返回位置值的字符搜索函数	216
12.2.17	字符包含函数	216
12.2.18	返回位置值的字符包含函数	217
12.2.19	在指定字符集中查找不包含字符函数	218
12.2.20	在指定字符集中查找包含字符函数	218
12.2.21	查找第一个包含字符函数	219
12.2.22	查找最后一个包含字符函数	219
12.3	I/O 函数	220
12.3.1	字符读入函数	220
12.3.2	字符读入输出函数	221
12.3.3	字符串读入函数	222
12.3.4	字符回送函数	222
12.3.5	字符输出函数	223
12.3.6	格式化输出函数	223
12.3.7	格式化内存缓冲区输出函数	225
12.3.8	字符串输出函数	226
12.3.9	格式化输入函数	227
12.3.10	格式化内存缓冲区输入函数	228
12.3.11	字符串内存输出函数	229
12.3.12	指向缓冲区的输出函数	230
12.4	数学函数	231
12.4.1	绝对值函数	231
12.4.2	指数及对数函数	232
12.4.3	三角函数	232
12.4.4	取整函数	234
12.4.5	浮点型分离函数	234
12.4.6	幂函数	235
12.5	标准函数	235
12.5.1	字符串转换函数	236
12.5.2	带返回指针的字符串转换函数	237
12.5.3	随机函数	238

12.5.4	数组内存分配函数	238
12.5.5	释放内存函数	239
12.5.6	初始化内存函数	240
12.5.7	内存分配函数	240
12.5.8	调整内存大小函数	241
12.6	内部函数	242
12.6.1	循环左移函数	242
12.6.2	循环右移函数	243
12.6.3	延时函数	244
12.6.4	位测试函数	244
12.7	绝对地址访问函数	245
12.7.1	BYTE 型存储空间访问函数	245
12.7.2	WORD 型存储空间访问函数	245
12.7.3	far 存储区访问函数	246
12.7.4	far 存储区数组访问函数	247
12.8	变量参数表函数	247
12.9	全程跳转函数	248
12.10	计算结构体成员的偏移量函数	249
12.11	小结	250

第 13 章 C51 语句和流程

251

本章详细讲述了单片机 C51 语言中的各种语句结构，包括说明语句、表达式语句、复合语句、循环语句、条件语句、开关语句、跳转语句、函数调用语句、空语句和返回语句。这些语句是 C51 语言程序的重要组成部分。灵活利用这些语句，可以实现不同的程序流程控制结构，如顺序结构、选择结构和循环结构等，以达到简化程序的目的。熟练掌握本章内容是进行 C51 语言程序设计的基础。

13.1	说明语句	251
13.2	表达式语句	251
13.3	复合语句	252
13.4	循环语句	253
13.4.1	while 语句	253
13.4.2	do-while 语句	254
13.4.3	for 语句	254
13.5	条件语句	255
13.5.1	单分支结构	255
13.5.2	双分支结构	255
13.5.3	阶梯式 if-else-if 结构	256
13.6	开关语句	258

13.7 跳转语句	259
13.7.1 goto 语句	259
13.7.2 break 语句	260
13.7.3 continue 语句	260
13.8 函数调用语句	261
13.9 空语句	261
13.10 返回语句	262
13.11 C51 语言的流程控制结构	263
13.12 小结	264

第 14 章 预处理及用户配置文件

265

本章详细介绍了 C51 语言所支持的各种预处理命令，包括宏定义指令、文件包含指令、条件编译指令和其他一些编译指令。然后还介绍了 C51 语言编译器的一些控制指令，这需要和具体的编译器相联系。最后还介绍了一下 C51 语言的用户配置文件。熟练掌握本章内容，对读者以后的单片机程序设计有很大帮助。

14.1 预处理命令概述	265
14.2 宏定义指令	265
14.2.1 #define 命令	266
14.2.2 #undef 命令	267
14.3 文件包含指令	268
14.4 条件编译指令	269
14.4.1 #if、#else、#endif 命令	269
14.4.2 #elif 命令	270
14.4.3 #ifdef、#ifndef 命令	270
14.5 其他编译指令	271
14.5.1 #line 命令	271
14.5.2 #error	272
14.5.3 #pragma	272
14.6 C51 语言编译器的控制指令	273
14.6.1 源文件控制类	273
14.6.2 目标文件 (Object) 控制类	273
14.6.3 列表文件 (Listing) 控制类	273
14.7 C51 语言的用户配置文件	274
14.7.1 C51 语言启动代码文件	274
14.7.2 C51 语言启动代码分析	277
14.7.3 变量初始化文件	279
14.7.4 基本 I/O 函数文件	279

14.7.5 分组配置文件	279
14.8 小结	280

第 15 章 C51 语言的存储结构

281

本章详细讲述了 C51 语言的存储器结构、存储类型、存储模式及存储器指针等，后面还介绍了动态内存分配。数据的存储模式是单片机系统特有的概念，这里的内容涉及单片机的硬件资源比较多，读者应该对照着单片机的介绍来进行学习，这样可以加深理解。

15.1 存储器结构	281
15.1.1 51 系列单片机的存储区域	281
15.1.2 片内数据存储器 (RAM) 的结构	281
15.2 存储类型	282
15.2.1 data 存储类型	282
15.2.2 bdata 存储类型	282
15.2.3 idata 存储类型	283
15.2.4 pdata 存储类型	283
15.2.5 xdata 存储类型	283
15.2.6 code 存储类型	284
15.3 扩展数据类型	284
15.3.1 sfr 和 sfr16	284
15.3.2 sbit	285
15.3.3 bit 型变量	285
15.4 存储模式	285
15.4.1 Small 模式	286
15.4.2 Compact 模式	286
15.4.3 Large 模式	286
15.4.4 存储模式的选择	286
15.5 C51 语言的存储器指针	286
15.5.1 一般指针	286
15.5.2 存储器指针	287
15.5.3 指针存储类型与指针所指向的数据的存储类型的关系	288
15.6 动态内存分配	288
15.6.1 C51 语言的动态分配函数	289
15.6.2 malloc 和 calloc 函数	289
15.7 小结	289

第四篇 51 系列单片机编程指南篇

本章详细讲解了 51 系列单片机的指令系统，包括指令的 7 种寻址方式，以及 51 系列单片机指令系统中的各类指令的书写格式、功能、使用方法及注意事项等。对于每一条指令，均给出了完整详细的实例来讲解如何在程序设计中应用。这一章的内容是读者学习使用单片机的基础必备知识，深刻地理解单片机指令系统，可以为接下来的学习打下良好的基础。

16.1	指令系统简介	290
16.1.1	指令格式	290
16.1.2	指令符号	291
16.2	寻址方式	292
16.2.1	立即寻址	292
16.2.2	直接寻址	293
16.2.3	寄存器寻址	295
16.2.4	寄存器间接寻址	295
16.2.5	变址寻址	297
16.2.6	相对寻址	298
16.2.7	位寻址	299
16.3	数据传送指令	300
16.3.1	内部 RAM 数据传送指令	300
16.3.2	外部 RAM 数据传送指令	304
16.3.3	程序存储器数据传送指令	305
16.3.4	数据交换指令	306
16.3.5	堆栈操作指令	308
16.4	算术运算指令	309
16.4.1	加法指令	309
16.4.2	带进位的加法指令	311
16.4.3	带借位的减法指令	312
16.4.4	加 1 指令	314
16.4.5	减 1 指令	315
16.4.6	乘除法指令	316
16.4.7	十进制调整指令	317
16.5	逻辑运算及移位指令	318
16.5.1	逻辑与指令	319
16.5.2	逻辑或指令	320
16.5.3	逻辑异或指令	321
16.5.4	累加器清零指令	322
16.5.5	累加器取反指令	323
16.5.6	组合逻辑电路的实现	324

16.5.7 循环移位指令	324
16.6 控制转移指令	326
16.6.1 无条件转移指令	326
16.6.2 条件转移指令	330
16.6.3 子程序调用及返回指令	334
16.7 位操作指令	337
16.7.1 位变量传送指令	337
16.7.2 置位与清零指令	338
16.7.3 位逻辑运算指令	339
16.7.4 位控制转移指令	340
16.8 空操作指令	342
16.9 51 系列单片机指令汇总	343
16.10 小结	347

第 17 章 51 系列单片机的定时器/计数器

348

本章详细讲述了 51 系列单片机的定时器/计数器的结构、控制寄存器及 4 种工作方式，并分别给出了详细的程序设计方法；本章还对 52 子系列的单片机定时器/计数器 T2 也进行了详细的介绍。定时器/计数器是单片机的一个非常有用的功能，熟练掌握本章内容，对读者以后的单片机设计有很大帮助。

17.1 定时器/计数器 0 和 1	348
17.1.1 定时器/计数器的结构	348
17.1.2 定时器/计数器的功能	349
17.1.3 T0 和 T1 的控制寄存器	349
17.2 定时器/计数器 0 和 1 的工作模式	352
17.2.1 工作模式 0 及其程序设计	352
17.2.2 工作模式 1 及其程序设计	354
17.2.3 工作模式 2 及其程序设计	356
17.2.4 工作模式 3 及其程序设计	358
17.3 定时器/计数器 2	360
17.3.1 T2 的控制寄存器 T2CON 和 T2MOD 及其程序访问	361
17.3.2 定时器/计数器 2 的工作模式	362
17.4 小结	367

第 18 章 51 系列单片机中断系统及其程序设计

368

本章详细讲述了中断系统的基本概念，并重点介绍了 51 系列单片机的中断类型及中断的各种控制标准位；接着讲述了 51 系列单片机对中断的处理过程，并通过实例详细讲述了各种中断源的编程方式；最后还介绍了外部中断源的扩展方式及其程序设计。中断是 51 系列单片机重要的系统资源，合理使用中断系统，可以减轻 CPU 的负担，简化程序设计，实现对外部信号的实时处理。因此，熟练掌握本章内容是学习 51 系列单片机的基础。

18.1	中断系统概述	368
18.1.1	什么是中断	368
18.1.2	中断的用途	368
18.1.3	中断需要解决的问题	369
18.2	51 系列单片机的中断类型	371
18.2.1	外部中断源	371
18.2.2	定时中断源	371
18.2.3	串行中断源	371
18.3	51 系列单片机的中断系统	372
18.3.1	中断请求标志及其访问	372
18.3.2	中断允许标志及其访问	373
18.3.3	中断优先级标志及其访问	374
18.4	中断的处理过程	375
18.4.1	中断响应	375
18.4.2	中断处理	377
18.4.3	中断返回	378
18.4.4	中断请求的撤离	378
18.5	中断源的程序设计	378
18.5.1	外部中断源的程序设计	378
18.5.2	定时中断源的程序设计	380
18.5.3	串行中断源的程序设计	381
18.6	外部中断源的扩展	382
18.6.1	定时器/计数器扩展外部中断源	382
18.6.2	查询方式扩展外部中断源	385
18.7	小结	386

第 19 章 51 系列单片机的串行接口

387

本章首先详细介绍了串行通信的基本方式，包括异步串行通信和同步串行通信，以及单工制式、半双工制式和全双工制式三种数据传送方式。51 系列单片机集成了全双工的串行接口，本章详细介绍了单片机串行接口的内部结构、程序控制、4 种工作模式及其程序设计等。最后介绍了单片机的串行接口在双机通信和多机通信方面的应用。单片机串行口的应用非常广泛，熟练掌握本章内容很重要。

19.1	串行通信概述	387
19.1.1	串行通信简介	387
19.1.2	串行通信的数据传送方式	389
19.2	51 系列单片机的串行接口	390
19.2.1	单片机串行接口的内部结构	390
19.2.2	单片机串行接口的程序控制	391

19.2.3	波特率的程序设计	393
19.3	串行口的工作模式 0	396
19.3.1	模式 0 的发送及扩展输出端口	396
19.3.2	模式 0 的接收及扩展输入端口	398
19.4	串行口的工作模式 1	399
19.4.1	模式 1 的发送	400
19.4.2	模式 1 的接收	401
19.5	串行口的工作模式 2	402
19.5.1	模式 2 的发送	402
19.5.2	模式 2 的接收	403
19.6	串行口的工作模式 3	405
19.6.1	模式 3 的发送	405
19.6.2	模式 3 的接收	406
19.7	双机通信程序设计	407
19.7.1	查询方式	407
19.7.2	中断方式	408
19.8	多机通信程序设计	410
19.8.1	多机通信原理	411
19.8.2	多机通信协议约定	411
19.8.3	多机通信程序设计	411
19.9	小结	416

第 20 章 C51 下的 RTX-51 实时多任务操作系统417

本章详细介绍了运行于 8051 硬件平台的 RTX-51 实时多任务操作系统。RTX-51 的程序不同于普通的单片机程序，这里对 RTX-51 的任务调度、系统函数、任务管理及 RTX-51 Tiny 的配置进行了详细的阐述。熟练掌握和运用本章内容，可以简化复杂的多任务单片机系统设计。

20.1	RTX-51 实时多任务操作系统简介	417
20.1.1	RTX-51 种类	417
20.1.2	RTX-51 与单任务程序的比较	418
20.2	RTX-51 的任务调度	419
20.2.1	RTX-51 循环任务调度	420
20.2.2	RTX-51 事件任务调度	420
20.2.3	RTX-51 信号任务调度	421
20.2.4	优先级及抢先任务切换	421
20.2.5	RTX-51 的其他特性	422
20.3	RTX-51 Tiny 的系统函数	423
20.3.1	发送信号函数 isr_send_signal	423

20.3.2	清除信号标志函数 <code>os_clear_signal</code>	423
20.3.3	启动任务函数 <code>os_create_task</code>	424
20.3.4	删除任务函数 <code>os_delete_task</code>	424
20.3.5	当前任务号函数 <code>os_running_task_id</code>	425
20.3.6	发送信号函数 <code>os_send_signal</code>	425
20.3.7	等待函数 <code>os_wait</code>	425
20.3.8	等待函数 <code>os_wait1</code>	426
20.3.9	等待函数 <code>os_wait2</code>	427
20.4	RTX-51 Tiny 的任务管理	428
20.4.1	RTX-51 Tiny 的任务状态	428
20.4.2	RTX-51 Tiny 的事件	428
20.4.3	RTX-51 Tiny 的任务切换	428
20.5	RTX-51 Tiny 的配置文件	429
20.6	RTX-51 Tiny 的要求及限定	431
20.6.1	使用 RTX-51 Tiny 的要求	431
20.6.2	RTX-51 Tiny 的注意事项	432
20.7	RTX-51 FULL 的系统函数及技术参数	433
20.7.1	RTX-51 FULL 函数一览	433
20.7.2	RTX-51 的技术参数	434
20.8	小结	435

第 21 章 Keil μ Vision3 中的单片机硬件资源仿真

436

本章详细介绍了 Keil μ Vision3 集成开发环境中，对单片机各种常见的片上资源的仿真操作。主要包括并行 I/O 端口、定时器/计数器、串行接口、中断、看门狗定时器、A/D、D/A、寄存器及低功耗仿真。单片机的程序设计主要是对各种片上资源进行操作，Keil μ Vision3 集成开发环境对各种片上资源均提供了完善的仿真支持。在程序设计时，通过仿真操作可以完美地模拟程序的执行情况，便于及时发现问题。这样便大大提高了程序开发的可靠性，加速了程序的开发速度。因此，读者应该熟练掌握本章内容。

21.1	仿真概述	436
21.2	并行 I/O 端口的仿真	436
21.3	定时器/计数器的仿真	438
21.3.1	定时器/计数器 T0 和 T1 的仿真界面	438
21.3.2	定时器/计数器 T2 的仿真界面	439
21.3.3	定时器/计数器的仿真操作	440
21.4	串行接口的仿真	442
21.4.1	串行接口的仿真界面	442
21.4.2	串行接口的仿真操作	443
21.4.3	字符串输入输出的仿真操作	444
21.5	中断仿真	446

21.5.1	中断系统的仿真界面	446
21.5.2	中断系统的仿真操作	447
21.6	看门狗定时器的仿真	448
21.6.1	看门狗定时器的仿真界面	448
21.6.2	看门狗定时器的仿真操作	448
21.7	A/D 仿真	449
21.7.1	A/D 转换器的仿真界面	450
21.7.2	A/D 转换器的仿真操作	450
21.8	D/A 仿真	452
21.8.1	D/A 转换器的仿真界面	452
21.8.2	D/A 转换器的仿真操作	453
21.9	寄存器仿真	455
21.9.1	寄存器的仿真界面	455
21.9.2	寄存器的仿真操作	455
21.10	低功耗仿真	456
21.11	小结	457

第 22 章 Keil μ Vision3 中的程序调试458

本章详细介绍了 Keil μ Vision3 集成开发环境的程序调试功能，包括性能分析器、代码覆盖分析器和断点等。本章还详细介绍了 Keil μ Vision3 编译器提供的调试命令，这些调试命令大大扩展了单片机程序的仿真调试。合理使用 Keil μ Vision3 的各种调试功能及调试命令，可以在程序设计和调试时达到事半功倍的效果。因此，熟练掌握本章内容有助于读者分析和优化单片机程序。

22.1	Keil μ Vision3 的程序调试器概述	458
22.2	性能分析器	458
22.3	代码覆盖分析器	459
22.4	断点	460
22.5	Keil μ Vision3 调试命令	461
22.5.1	通用命令	461
22.5.2	程序命令	465
22.5.3	断点命令	467
22.5.4	存储器命令	469
22.6	小结	471

第五篇 典型案例篇

第 23 章 键盘程序设计472

本章详细讲述了键盘设计需要注意的一些问题，然后介绍了独立式按键和矩阵式键盘的工作方式。

其中着重讲了矩阵式键盘的扫描法、线反转法和中断法的工作原理及程序设计。最后通过一个完整电路，实现了矩阵式键盘的扫描查询方式在程序中的应用。在实例中，还通过仿真分析了整个程序流程的正确性。矩阵式键盘应用十分广泛，熟练掌握它的使用是学习单片机应用系统的基础。

23.1	键盘接口概述	472
23.1.1	按键编码	472
23.1.2	输入的可靠性	472
23.1.3	程序检测及响应	473
23.2	独立式按键及其编程接口	473
23.2.1	独立式按键结构	473
23.2.2	独立式按键程序设计	474
23.3	4×4 矩阵式键盘及其编程接口	475
23.3.1	扫描法及其程序设计	475
23.3.2	线反转法及其程序设计	477
23.3.3	中断法及其程序设计	479
23.4	矩阵式键盘的接口实例	480
23.4.1	电路图	480
23.4.2	程序设计	481
23.4.3	程序仿真	483
23.5	小结	485

第 24 章 LED 数码管显示

486

本章详细介绍了 LED 数码管显示器件，包括 7 段共阳极 LED 数码管和 7 段共阴极 LED 数码管，然后介绍了 LED 的静态显示技术及其应用实例。本章还重点讲解了 LED 数码管的动态显示技术，包括静态驱动、动态驱动和 LED 驱动器驱动。最后通过一个具体的实例讲解了使用 LED 驱动器控制多个 LED 的显示。LED 数码管显示是单片机系统中常用的显示接口，读者应该熟练掌握其使用方法。

24.1	LED 数码管概述	486
24.1.1	7 段共阳极 LED 结构及显示段码	486
24.1.2	7 段共阴极 LED 结构及显示段码	487
24.2	单个 LED 驱动实例	488
24.2.1	电路图	488
24.2.2	程序设计	490
24.3	多个 LED 驱动方式	492
24.3.1	静态驱动显示	493
24.3.2	动态驱动显示	497
24.3.3	LED 驱动器	499
24.4	多个 LED 驱动实例	503
24.4.1	LED 驱动器电路图	503
24.4.2	程序设计	504

24.5 小结	508
---------------	-----

第 25 章 LCD 液晶显示模块

509

本章详细介绍了液晶和液晶显示模块的结构及原理，并对常用的 LCD 驱动控制器的指令和功能进行了介绍，其中给出了用于图形点阵式液晶读写的详细子函数。最后利用一款采用这个驱动器的液晶显示模块，来介绍如何控制其显示汉字和图形等。近年来，液晶显示模块应用越来越广泛。熟练掌握本章内容对读者以后的设计很有帮助。

25.1 LCD 液晶显示概述	509
25.1.1 液晶的来源	509
25.1.2 LCD 液晶显示器结构及原理	509
25.1.3 液晶显示模块的种类	510
25.1.4 液晶显示模块的优点	511
25.2 液晶显示模块控制接口	511
25.2.1 LCD 控制驱动器 ST7920 概述	511
25.2.2 ST7920 功能说明	512
25.2.3 ST7920 基本指令集	516
25.2.4 ST7920 扩充指令集	518
25.2.5 ST7920 的操作方式	520
25.2.6 图形点阵式液晶显示模块	521
25.2.7 图形点阵式液晶读写子函数	522
25.3 汉字及图形显示实例	525
25.3.1 电路设计	525
25.3.2 建立项目	526
25.3.3 汉字显示实例	526
25.3.4 图形显示实例	528
25.3.5 任意位置图形显示实例	530
25.4 小结	532

第 26 章 D/A 转换实例

533

本章首先详细介绍了 D/A 转换器的基本知识，包括 D/A 转换器的原理、D/A 转换器的类型及技术参数，然后介绍了一个高速易使用的 D/A 转换芯片 AD558。最后通过一个具体的实例，介绍了如何采用 AD558 在光通信领域中完成对光信号的相位精确调制。D/A 转换器在测控领域有着广泛的应用，扩展了 8051 单片机处理模拟信号的能力，是现代电子设计中不可缺少的一部分。

26.1 D/A 转换概述	533
26.1.1 D/A 转换原理	533
26.1.2 D/A 转换器的类型	534
26.1.3 D/A 转换器的技术参数	535
26.2 高速 D/A 转换芯片 AD558	536

26.2.1	AD558 简介	536
26.2.2	AD558 电压输出模式	537
26.2.3	AD558 的数据锁存	538
26.3	光通信电压调制电路实例——电路部分	539
26.3.1	相位调制的原理	539
26.3.2	电压调制系统	540
26.3.3	电路图	541
26.4	光通信电压调制电路实例——程序部分	544
26.4.1	系统状态编码	544
26.4.2	建立项目	544
26.4.3	主程序	545
26.4.4	无调制模式函数	546
26.4.5	调制模式 1 函数	546
26.4.6	调制模式 2 函数	547
26.4.7	调制模式 3 函数	547
26.4.8	调制模式 4 函数	547
26.5	光通信电压调制电路实例——仿真部分	547
26.5.1	程序仿真	548
26.5.2	运行效果	548
26.6	小结	548

第 27 章 可编程逻辑器件 CPLD

549

本章首先介绍了可编程逻辑器件的发展，以及 CPLD 和 FPGA 的结构及逻辑实现。可编程逻辑器件一般采用 VHDL 语言进行设计，本章对 VHDL 语言进行了简要的介绍。本章对常用的 CPLD 芯片 EPM7128SLC84 进行展开讲解，提供了 Altera 公司的 CPLD 下载电路。本章实例部分，使用 CPLD 配合 AT89S52 来扩展了 8051 单片机的并行 I/O 端口。单片机和可编程逻辑器件内部结构不同，各有优势，实际系统中经常需要将两者结合使用。

27.1	可编程逻辑器件概述	549
27.1.1	可编程逻辑器件的发展	549
27.1.2	CPLD 的结构及其逻辑实现	549
27.1.3	FPGA 的结构及其逻辑实现	551
27.2	硬件描述语言简述	552
27.2.1	硬件描述语言 VHDL 概述	553
27.2.2	VHDL 程序结构	553
27.3	Altera 常用 CPLD 芯片介绍	555
27.4	使用 CPLD 扩展 51 单片机 I/O 接口	557
27.4.1	CPLD 扩展单片机 I/O 接口原理	558
27.4.2	电路图	558

27.5 单片机程序设计559

27.5.1 项目建立559

27.5.2 主程序560

27.6 VHDL 程序设计560

27.6.1 项目建立560

27.6.2 程序设计561

27.7 程序仿真563

27.7.1 设计 CPLD 引脚563

27.7.2 仿真操作564

27.8 程序下载565

27.9 小结567

第 28 章 51 系列单片机读写 I2C 总线

568

本章详细介绍了 I2C 串行总线的工作原理、结构及寻址方式等，并对 I2C 串行总线的数据传输进行了详细的介绍。本章还给出了采用普通的 51 系列单片机模拟读写 I2C 串行总线的汇编语言和 C 语言的代码。最后通过一个具体的实例，讲解了单片机读写 I2C 总线外围器件的电路设计及程序设计。I2C 串行总线具有接口简单，体积小等优点，在实际电路设计中经常使用。熟练掌握本章内容，可以控制大部分的 I2C 总线外围器件，从而大大扩展了 51 系列单片机的使用范围。

28.1 I2C 总线概述568

28.1.1 I2C 总线工作原理568

28.1.2 I2C 总线的电气结构和负载能力569

28.1.3 I2C 总线器件的寻址方式569

28.2 I2C 总线数据传输协议及其程序详解570

28.2.1 起始信号570

28.2.2 终止信号571

28.2.3 应答信号571

28.2.4 非应答信号572

28.2.5 应答位检查573

28.2.6 总线数据位573

28.2.7 写数据573

28.2.8 读数据576

28.3 51 单片机读写 EEPROM579

28.3.1 串行 EEPROM 存储器简介579

28.3.2 电路设计580

28.3.3 程序设计581

28.3.4 仿真分析582

28.4 小结583

第 29 章 单片机音乐播放

584

本章详细讲述了音乐学中音调和节拍的概念，以及如何使用单片机来实现音调和节拍的演奏。本章还给出了一些音乐片段的示例代码。最后通过一个具体的实例，讲解了如何使用单片机播放音乐。单片机的功能强大，成本低廉，对于需要简单音乐播放的场合，可以选择使用单片机控制扬声器来实现。

29.1 单片机发音概述	584
29.1.1 音调	584
29.1.2 节拍	586
29.1.3 单片机音乐播放的方法及音乐示例	586
29.2 单片机音乐播放实例——电路图	589
29.3 单片机音乐播放实例——程序设计	590
29.3.1 建立项目	590
29.3.2 程序设计	591
29.4 小结	593

第 30 章 实时时钟芯片应用

594

本章详细介绍了实时时钟芯片 DS1302 的命令字节和数据格式，并介绍了两种数据传输方式，即单字节传输方式和多字节突发传输方式。本章通过一个具体的实例，讲解了如何使用 8051 单片机来实现对 DS1302 的控制，在该实例中分别采用了单字节传输方式和多字节传输方式来对 DS1302 的时钟寄存器及 RAM 进行操作。实时时钟常用于需要时间设定和显示的场合，在单片机应用系统中很常见，读者熟练掌握本章内容，可以轻松实现单片机的时钟显示扩展。

30.1 实时时钟芯片 DS1302 概述	594
30.1.1 实时时钟芯片 DS1302 概述	594
30.1.2 实时时钟芯片 DS1302 命令字节	595
30.1.3 实时时钟芯片 DS1302 数据格式	595
30.1.4 实时时钟芯片 DS1302 数据传输方式	597
30.2 单片机读写实时时钟芯片实例	598
30.2.1 电路图	598
30.2.2 建立项目	598
30.2.3 主程序	599
30.2.4 复位函数	601
30.2.5 字节读取函数	602
30.2.6 字节写入函数	602
30.2.7 初始化函数	602
30.2.8 时钟字节写入函数	603
30.2.9 RAM 字节写入函数	604
30.2.10 时钟寄存器内容显示函数	604
30.2.11 多字节突发方式读取 RAM 函数	605

30.2.12 多字节突发方式写入 RAM 函数	605
30.3 小结	606

第 31 章 静态 RAM 存储器应用607

本章详细讲述了 RAM 存储器的种类和特点，并重点讲解了应用最为广泛的静态 RAM 存储器。本章还对一款常用的静态 RAM 存储器 HM628128 进行了详细介绍。最后，通过一个综合的实例介绍了静态 RAM 存储器的读写。实例中使用了计算机的串行通信接口，以及单片机的串行接口设计。通过本章的讲解，读者可以掌握单片机系统中静态 RAM 存储器的读写及计算机串行的应用。

31.1 静态 RAM 存储器概述	607
31.1.1 RAM 存储器概述	607
31.1.2 静态 RAM 芯片 HM628128	608
31.1.3 静态 RAM 芯片 HM628128 的读写	608
31.2 静态 RAM 存储器读写实例	609
31.2.1 系统原理	610
31.2.2 串行通信接口概述	610
31.2.3 单片机与 RS-232C 的接口	615
31.2.4 系统电路原理图	616
31.2.5 建立项目	617
31.2.6 主程序设计	618
31.2.7 系统运行	620
31.3 小结	622

第 32 章 道路交通灯控制系统623

本章详细讲述了道路交通灯的运行原理，以及如何使用 RTX-51 Tiny 程序来实现道路交通灯的控制模拟。本章给出了详细的电路图，以及 RTX-51 Tiny 的多任务程序。通过本章的学习，可以掌握实时多任务操作系统的设计，尤其是基于 8051 单片机的 RTX-51 Tiny 的程序设计。

32.1 交通灯控制系统概述	623
32.1.1 道路交通灯概述	623
32.1.2 交通灯控制系统	623
32.2 交通灯控制系统原理图	624
32.3 多任务交通灯控制系统程序	625
32.3.1 建立项目	625
32.3.2 多任务划分及程序设计	626
32.3.3 串行通信函数	632
32.3.4 获取命令函数	635
32.4 小结	636

第 33 章 单总线温度传感器 DS18S20637

本章介绍了 1-Wire 单总线的工作原理,并结合 1-Wire 总线接口温度传感器 DS18S20,详细讲解了其供电方式及数据操作命令。最后通过一个完整的实例介绍了如何使用 51 系列单片机模拟 1-Wire 总线数据传输,从而实现 DS18S20 的控制。1-Wire 单总线是一种结构简单的接口协议,其最大化地减少了 I/O 引脚数目,在实际电路中有着广泛的应用。

33.1 单总线概述	637
33.2 单总线温度传感器 DS18S20	638
33.2.1 温度传感器 DS18S20 概述	638
33.2.2 DS18S20 的供电方式	639
33.2.3 DS18S20 的数据操作	639
33.3 单片机读写温度传感器 DS18S20 实例	642
33.3.1 电路图	642
33.3.2 建立项目	643
33.3.3 DS18S20 读写子函数	643
33.3.4 主函数	649
33.3.5 程序仿真	651
33.4 小结	651

第 34 章 Microware 串行总线 EEPROM 的应用

652

本章详细讲解了三线制 Microware 串行总线,并通过 Microware 串行总线的 EEPROM 介绍了 Microware 串行总线的操作指令及其操作时序。本章最后通过一个具体的实例,介绍了如何使用标准的 51 系列单片机来在软件上仿真模拟 Microware 串行总线。其中给出了 Microware 串行总线读写的子函数及一个完整的实例。三线制 Microware 串行总线减少了 I/O 引脚的使用,在实际电路设计中具有广泛的应用,读者应该熟练掌握。

34.1 三线制 Microware 串行总线概述	652
34.2 Microware 串行总线接口的 EEPROM	653
34.2.1 Microware 串行总线接口 EEPROM 概述	653
34.2.2 Microware 串行总线接口 EEPROM 的指令	653
34.2.3 Microware 串行总线接口 EEPROM 的指令时序	654
34.3 51 系列单片机读写三线制 EEPROM 实例	657
34.3.1 电路图	657
34.3.2 建立项目	657
34.3.3 三线制 Microware 串行总线读写子函数	658
34.3.4 主程序	660
34.3.5 Microware 串行总线仿真	662
34.4 小结	663

第 35 章 单片机控制打印机实例

664

本章详细讲述了 LASER PP40 微型四色描绘式打印机,包括其接口、工作时序、文本模式和图形模

式等。本章最后还通过一个具体的实例，介绍了如何使用 51 系列单片机控制 LASER PP40 微型打印机打印输出。LASER PP40 微型打印机接口简单、控制方便，十分适合作为单片机系统的智能输出扩展。因此，熟练掌握本章内容对读者以后的单片机系统设计工作很有帮助。

35.1 打印机概述	664
35.1.1 LASER PP40 打印机概述	664
35.1.2 LASER PP40 的文本模式	665
35.1.3 LASER PP40 的图形模式	666
35.2 51 系列单片机控制打印机实例	667
35.2.1 电路图	667
35.2.2 建立项目	668
35.2.3 程序设计	668
35.3 小结	669

第 36 章 A/D 转换实例

670

本章首先详细介绍了 A/D 转换的原理、A/D 转换器的技术参数及 A/D 转换器的选用原则。接着，本章介绍了一个高性能的 8 通道 A/D 转换器 MAX197。其中，详细讲解了 MAX197 的特性、引脚功能，以及接口、控制和时序逻辑等。最后，本章通过一个完整的实例介绍了 A/D 转换器 MAX197 与单片机的接口。A/D 转换在测控领域，特别是模拟信号的数据采集系统中有着广泛的应用，读者应熟练掌握 A/D 转换的相关知识及 A/D 转换器的使用。

36.1 A/D 转换概述	670
36.1.1 A/D 转换原理	670
36.1.2 A/D 转换器的技术参数	672
36.1.3 A/D 转换器的选择原则	673
36.2 8 通道 A/D 转换器 MAX197	673
36.2.1 MAX197 的特性及引脚功能	674
36.2.2 MAX197 的接口、控制字及时序	675
36.3 单片机读写 A/D 转换器实例	677
36.3.1 电路图	677
36.3.2 建立项目	679
36.3.3 程序设计	679
36.3.4 程序仿真	682
36.4 小结	682

第 37 章 单片机读写智能 IC 卡

683

本章主要介绍了智能 IC 卡的相关知识。其中对目前市场上广泛使用的 AT45DB041D 接触式 IC 卡芯片进行了详细介绍，包括芯片的功能、内存空间和指令。本章还通过一个具体的实例，介绍了如何使用单片机对 IC 卡芯片进行读写操作。由于 IC 卡芯片采用 SPI 串行数据接口，这里使用了带有 SPI 接口的 AT89S8253 单片机进行读写操作。智能 IC 卡目前得到广泛的使用，读者应该熟练掌握本章内容。

37.1 智能 IC 卡概述683

37.1.1 智能 IC 卡分类.....683

37.1.2 接触式 IC 卡684

37.1.3 非接触式 IC 卡.....684

37.2 智能 IC 卡芯片684

37.2.1 IC 卡芯片 AT45DB041 简介684

37.2.2 AT45DB041D 的内存空间及其读写.....685

37.2.3 AT45DB041D 的指令686

37.3 单片机读写智能 IC 卡实例687

37.3.1 电路图687

37.3.2 SPI 接口单片机 AT89S8253 简介.....688

37.3.3 建立项目689

37.3.4 IC 卡芯片 AT45DB041D 读写子函数.....690

37.3.5 主程序692

37.4 小结.....694

第 38 章 单片机智能锂电池充电管理

695

本章首先介绍了广泛使用的锂电池，以及锂电池的充电要求。接着介绍了 MAXIM 公司的一款高性能的智能充电管理芯片 MAX1898，包括 MAX1898 引脚功能及其充电工作原理。最后，本章通过一个具体的实例，介绍了如何使用 51 系列单片机控制 MAX1898 来实现单节锂电池的智能充电过程。锂电池及其充电器广泛应用于生活中，因此，熟练掌握本章内容具有极大的实际意义。

38.1 锂电池及其充电概述695

38.1.1 锂电池概述695

38.1.2 锂电池充电概述.....696

38.2 智能充电管理芯片 MAX1898.....696

38.2.1 智能充电管理芯片 MAX1898 概述.....696

38.2.2 MAX1898 充电工作原理697

38.3 单片机智能控制锂电池充电实例698

38.3.1 电路图699

38.3.2 智能充电器的功能.....700

38.3.3 建立项目700

38.3.4 程序设计701

38.4 小结.....702

第一篇 51 系列单片机基础

第 1 章 51 系列单片机概述

单片机以其价格低廉、功能强大、体积小、性能稳定等优点，深受广大电子设计爱好者喜爱。目前，各种产品中都能看到单片机的身影，如门铃、报警器、玩具，以及各种数据采集系统和控制终端等。单片机是现代电子设计中使用最为广泛的电子元件之一，而其中的 51 系列单片机是最早兴起的一类。51 系列单片机功能完备、指令系统丰富，发展最为成熟。

本章主要介绍单片机的产生及几十年的发展演化、51 系列单片机的简介和应用领域；另外，本章还包括最新主流 51 内核单片机介绍及单片机的开发概述。

1.1 单片机的产生与发展

1946 年，美国宾夕法尼亚大学研制成功了世界上第一台电子数字计算机 ENIAC。该计算机运算速度为 5000 次/s，内部使用了 18000 多个电子管和 1500 多个继电器，占地面积 150m²，重约 30 吨。它的诞生引发了 20 世纪电子工业的革命，如今电子计算机以令人难以想象的速度发展，产品线不断更新换代，成为当前发展最快的行业。

近年来，为了满足小型设备或便携式设备的需求，在计算机的大家族中，单片微型计算机异军突起，发展十分迅速，其基本渗透到了电子设计领域的各个方面。

单片微型计算机（Single-Chip Microcomputer）是在一块芯片上集成了中央处理器（Central Processing Unit, CPU）、只读存储器（Read Only Memory, ROM）、随机存取存储器（Random Access Memory, RAM）、定时器/计数器及 I/O（Input/Output）接口等部件，这些部件构成了一个完整的微型计算机。单片微型计算机简称单片机。单片机从产生到现在的短短三十几年历史中，产品不断更新，其发展大致经历了 4 个阶段。

1. 4 位单片机时代

第一阶段是 4 位单片机时代（1970 年—1974 年），这时的单片机已经包含多种 I/O 接口，如并行接口、A/D 和 D/A 转换接口等。这些丰富的 I/O 接口使得 4 位单片机具有很强的控制能力。其主要用于收音机、电视机和电子玩具等产品中。

2. 低中档 8 位单片机时代

第二阶段是中档 8 位单片机时代（1974 年—1978 年），Intel 公司的 MCS-48 系列单片机是其中主要的代表产品。这时的单片机内部集成了 8 位 CPU、多个并行 I/O 口、8 位定时器/计数器、小容量的 RAM 和 ROM 等。这种单片机中没有集成串行接口，操作仍比较简单。

3. 高档 8 位单片机时代

第三阶段是高档 8 位单片机时代（1978 年—1983 年），以 Intel 公司的 MCS-51 系列为典型代表。此时的单片机性能比前一代产品有明显提高，其内部增加了串行通信接口，具备多级中

51 单片机开发与应用技术详解

断处理系统，将定时器/计数器扩展为 16 位，并且扩大了 RAM 和 ROM 的容量等。这类单片机功能强，应用范围极广，至今仍有一定的应用市场。

4. 增强型单片机时代

第四阶段是增强型单片机时代及 16 位单片机时代（1983 年—现在）。这一阶段出现了许多新型的 8 位增强型单片机，其工作频率、内部存储器等都有很大的提升，例如 PIC 系列单片机、ARM 系列单片机、AVR 系列单片、C8051F 系列单片机等。另外有些集成电路厂商还推出了 16 位单片机，甚至 32 位单片机，其功能越来越强大，集成度越来越高。

总的来说，现在的单片机产品非常丰富，但 4 位、8 位、16 位单片机均有其各自的应用领域。例如 4 位单片机在一些简单的家电和玩具中使用，8 位单片机在中、小规模电子设计领域中占主流，而高性能的 16 位单片机在比较复杂的控制系统中得到应用。

1.2 51 系列单片机介绍

目前 8 位单片机仍然是电子设计领域使用最为广泛的产品，这里将详细介绍使用最多的 51 系列 8 位单片机。

1.2.1 51 系列单片机简介

51 系列单片机是指 Intel 的 MCS-51 系列及和其具有兼容内核的单片机。MCS-51 系列单片机是最早、最基本的单片机，功能也最简单。Intel 公司生产的 MCS-51 系列单片机包括 8031、8051、8032、8052、8751、8752 等。

现在集成电路技术飞速发展，各大芯片厂商提供了很多与 MCS-51 兼容的单片机。比如 Atmel 公司的 AT89C 系列、AT89S 系列，Silicon Laboratories 公司的 C8051F 系列，还有 Philips 公司的 8XC552 系列等。这些单片机采用兼容 MCS-51 的结构和指令系统，只是对其功能和内部资源等方面进行了不同程度的扩展。

这些具有兼容内核的 51 系列单片机，由于硬件结构和指令系统的一致性，大大方便了程序的移植，以及系统的升级，使得其使用起来十分方便。

除了 51 系列单片机以外，还有其他一些指令集和内部结构不同的单片机，它们与 51 系列单片机一般不兼容。这些单片机包括 PIC 单片机、ARM 系列单片机等。

1.2.2 51 系列单片机的应用领域

51 系列单片机以其高性能、高速度、体积小、价格低廉、可重复编程和方便功能扩展等优点，在市场上得到广泛的应用。其主要应用于如下几个领域。

- 家电产品及玩具。由于 51 系列单片机价格低、体积小、控制能力强、功能扩展方便等优点，使其广泛应用于电视、冰箱、洗衣机、玩具、家用防盗报警器等方面。
- 机电一体化设备。机电一体化设备是指将机械技术、微电子技术和计算机技术结合在一起，从而产生具有智能化特性的产品，它是现代机械及电子工业的主要发展方向。单片机可以作为机电一体化产品的控制器，从而简化原机械产品的结构，并扩展其功能。
- 智能测量设备。以前的测量仪表体积大、功能单一，限制了测量仪表的发展。采用单片机改造各种测量控制仪表，可以使其体积减少、功能扩展，从而产生新一代的智能化仪表，如各种数字万用表、示波器等。
- 自动测控系统。采用单片机可以设计各种数据采集系统、自适应控制系统等。例如温度的自动控制、电压电流的数据采集。
- 计算机控制及通信技术。51 系列单片机都集成有串行通信接口，可以通过该接口和计

算机的串行接口进行通信，实现计算机的程序控制和通信等。

1.3 最新 51 内核单片机介绍

自世界上第一片单片机诞生以来，51 系列单片机不断推陈出新，目前已有几十个系列、上百种型号。这些新产品都基于 51 内核，各个型号之间基本都兼容。以下是一些典型的 51 系列单片机。

- 美国 Intel 公司的 MCS-48 系列、MCS-51 系列、MCS-96 系列单片机；
- 美国 Atmel 公司的 AT89 系列单片机；
- 美国 Motorola 公司的 6801、6802、6803、6805 和 68HC11 系列单片机；
- 美国 Zilog 公司的 Z8、Super8 系列单片机；
- 美国 Fairchild 公司的 F8 和 3870 系列单片机；
- 美国 TI 公司的 TMS7000 系列单片机；
- 美国 NS 公司的 NS8070 系列单片机；
- 日本 NEC 公司的 μ PD7800 系列单片机；
- 日本 Hitachi 公司的 HD6301、HD6305 系列单片机。

最近几年，随着半导体技术的发展，不同厂商对各自的 51 系列单片机的功能进行了增强，包括执行速度、内部资源、电源系统及指令系统等。这里主要介绍一下目前应用比较广、影响比较大的一些 51 内核的单片机。这些单片机性能优越，推荐读者在自己的设计中采用。

1.3.1 Atmel 单片机介绍

Atmel 公司的产品非常丰富，除了基本的 51 系列单片机外，还包括针对不同设计领域的专用 51 内核单片机。Atmel 公司的 51 内核单片机有如下几类。

- 单周期 8051 内核单片机。这类单片机具有单周期 8051 内核，Flash ISP 在系统编程调试，片内集成了 SPI、UART、模拟比较器、PWM 及内部 RC 振荡器等资源。主要有 AT89LP213、AT89LP214、AT89LP216、AT89LP2052、AT89LP4052 等。
- Flash ISP 在系统编程单片机。这类单片机主要特点是内部集成 Flash，可以实现 ISP 在系统编程，使用方便。包括 AT89C5115、AT89C51AC2、AT89C51AC3、AT89C51ED2、AT89C51IC2、AT89C51ID2、AT89C51RB2、AT89C51RC2、AT89C51RD2、AT89C51RE2、AT89LS51、AT89LS52、AT89S2051、AT89S4051、AT89S51、AT89S52、AT89S8253 等。
- USB 接口单片机。这类单片机片内集成 USB 接口，基于 C51 微处理器，另外还具备 TWI、SPI、UART、PCA、ADC 等资源。包括 AT83C5134、AT83C5135、AT83C5136、AT89C5130A-M、AT89C5131A-L、AT89C5131A-M、AT89C5132 等。
- 智能卡接口单片机。这类单片机基于 C51 微处理器，带有串行接口和智能卡接口、DC/DC 转换，以及 EEPROM 等资源。包括 AT83C5121、AT83C5122、AT83C5123、AT83C5127、AT83EC5123、AT85C5121、AT85C5122、AT85EC5122、AT89C5121 等。
- MP3 专用单片机。这类单片机基于 C51 内核，具备 USB、多媒体卡接口、ADC、DAC、TWI、UART、SPI、MP3、WMA、JPEG 及 MPEG 的编解码电路等。包括 AT85C51SND3、AT89C51SND2C、AT83SND2C、AT89C51SND1C、AT83SND1C、AT80C51SND1C 等。

1.3.2 Cypress 单片机介绍

Cypress 公司的 51 内核单片机主要集中在 USB 接口上，有如下几类。

- USB 嵌入式主机。包括 CY7C67200、CY7C67300、SL811HST 等。

51 单片机开发与应用技术详解

- USB 全速设备。包括 AN21xx 系列、CY7C64013C、CY7C64215、CY7C6431x 系列、CY7C64345、CY7C6435x 系列、CY7C64713 等。
- USB 高速设备。包括 CY7C68001、CY7C68013A、CY7C68014A、CY7C68015A、CY7C68016A、CY7C68023、CY7C68024、CY7C68033、CY7C68034 等。
- USB 低速设备。包括 CY7C630xx、CY7C631xx、CY7C632xx、CY7C633xx、CY7C63413C、CY7C63513C、CY7C63613C、CY7C637xx、CY7C638xx 等。

1.3.3 Infineon 单片机介绍

Infineon 公司的产品包括标准的 8051 内核及符合工业标准的 8051 单片机。主要有如下几类。

- XC800 系列单片机。新型高级 XC800 家族 8 位微控制器采用高性能 8051 内核、片上集成闪存和 ROM 存储器及功能强大的外设组，如增强型 CAPCOM6 (CC6)、CAN、LIN 和 10 位 ADC，具有多种产品型号可供选择。如 XC886/888CLM、XC886/888LM、XC866 等。
- C500/C800 系列单片机。这类单片机是基于工业标准 8051 架构的微处理器，具有 CAN、SPI 等资源。包括 C515C、C505CA、C868 等。

1.3.4 Silicon 单片机介绍

Silicon Laboratories 公司的 C8051F 系列单片机，集成了世界一流的模拟功能、Flash 及基于 JTAG 的调试功能。另外还具有可配置的高性能模拟、高达 100 MIPS 的 8051CPU 及系统内现场可编程性，这些特性为用户提供了充分的设计灵活性及卓越的系统性能。C8051F 系列单片机主要有如下几类。

- USB 混合信号微处理器。这类微处理器内部集成了 USB 接口，以及 ADC、DAC、温度传感器、SMbus、UART 等资源。包括 C8051F340、C8051F341、C8051F342、C8051F343、C8051F344、C8051F345、C8051F346、C8051F320、C8051F321、C8051F326、C8051F327 等。
- 精密混合信号微处理器。这类微处理器内部集成了 Flash、ADC、DAC、温度传感器、SMbus、UART、比较器、VREF 等资源。包括 C8051F120、C8051F121、C8051F122、C8051F123、C8051F124、C8051F125、C8051F126、C8051F127、C8051F130、C8051F131、C8051F132、C8051F133、C8051F350、C8051F351、C8051F352、C8051F353、C8051F020、C8051F021、C8051F022、C8051F023、C8051F064、C8051F065、C8051F066、C8051F067、C8051F001、C8051F002、C8051F005、C8051F006、C8051F007、C8051F010、C8051F011、C8051F012、C8051F015、C8051F016、C8051F017、C8051F018、C8051F019 等。
- CAN 总线接口混合信号微处理器。这类微处理器内部集成了 CAN 总线接口、Flash、ADC、DAC、温度传感器、SMbus、UART、比较器、VREF 等资源。包括 C8051F040、C8051F041、C8051F042、C8051F043、C8051F044、C8051F045、C8051F046、C8051F047、C8051F060、C8051F061、C8051F062、C8051F063 等。
- 小型化微处理器。这类微处理器将高速 8051 CPU、闪存及高性能模拟电路集成到一个超小微型导线框封装 (MLP) 中，可以让系统设计者在提高系统性能的同时，减少元件数量和整体尺寸。包括 C8051F360、C8051F361、C8051F362、C8051F363、C8051F364、C8051F365、C8051F410、C8051F411、C8051F412、C8051F413、C8051F310、C8051F311、C8051F316、C8051F317、C8051F206、C8051F226、C8051F236、C8051T600、C8051T601、C8051F526、C8051F527 等。

1.3.5 Maxim 单片机介绍

Maxim 公司的产品很丰富，其推出的 8051 兼容微控制器在保持指令集、目标代码与早期 8051 设计兼容的同时，使性能指标提高 33 倍。主要有如下几类。

- 高速微处理器。这类微处理器具有闪存、EPROM、ROM 等，每机器周期使用一个时钟，速度是标准 8051 的 33 倍。包括 DS89C450、DS89C430、DS87C530、DS87C520、DS83C530、DS83C520、DS80CH11、DS80C323、DS80C320、DS80C310 等。
- 安全微控制器。这是具有防篡改能力的微控制器，其能够对程序和数据存储器进行加密，以防止未经授权的系统介入。系统的电池备份架构一旦检测到篡改事件将立即“清零”内部 SRAM，并且 DES/3DES 加密技术可以防止外部总线窃听。包括 DS5250、DS5000T、DS5000、DS2250T、DS2250、DS5002FP、DS2252T、DS907X、DS5001FP、DS5000FP、DS2251T 等。
- 网络微控制器。Maxim 的微型互联网接口（TINI）网络微控制器能够为嵌入式系统增添网络功能，适用于以太网或各种低级网络系统。片内集成具有 IPv4/IPv6 的 TCP/IP 网络栈，以及 10/100 以太网 MAC，符合 IEEE® 802.3 MII 标准。包括 DS80C411、DS80C410、DS80C400、DS80C390 等。

1.3.6 NXP 单片机介绍

NXP 半导体公司的前身是 Philips，其推出了多种单片机微控制器。主要有如下几类。

- LPC7000 系列。主要有 P87LPC760、P87LPC761、P87LPC762、P87LPC764、P87LPC767、P87LPC768、P87LPC769、P87LPC778、P87LPC779 等。
- LPC9000 系列。这是一种增强型多用途 Flash 单片机。主要有 P89LPC9401、P89LPC9402、P89LPC9403、P89LPC9408、P89LPC9102、P89LPC9103、P89LPC9107、P89LPC912、P89LPC913、P89LPC914、P89LPC915 及 P89LPC92x 系列、P89LPC93x 系列等。
- 80C51 系列。包括 P87C5xX2、P87CL5xX2、P89C5xX2、P89C66x、P8xC591、P87C552、P87C5x、P89C5xBx、P87C51Rx 等。

1.3.7 Winbond 单片机介绍

Winbond 系列单片机是中国台湾的华邦电子推出的，其产品丰富。主要有如下几类。

- 标准 51 单片机。这类单片机具有高达 40MHz 的工作频率，包含多个定时/计数器及在系统编程等特性。包括 W78C32、W78E51B、W78E52B、W78E54B、W78E58B、W78E516、W78E858、W78C51D、W78C52D、W78C54、W78C801、W78C438C、W78C58 等。
- 宽电压单片机。这类单片机工作电压可以低至 2.4V 及 1.8V，非常适合于电池供电的手持式设备。包括 W78L32、W78L51、W78L52、W78L54、W78L801、W78LE51、W78LE52、W78LE54、W78LE58、W78LE516、W78LE812 等。
- 增强 C51 单片机。这类单片机工作电压可以低至 2.7V，具有高达 40MHz 的工作频率、多个定时/计数器、12 个中断源、内置 SRAM，以及双 UART 等资源。主要包括 W77C32、W77L32、W77E58、W77LE58 等。
- 工业温度级单片机。这类单片机具有符合工业应用的温度范围及低至 2.4V 的工作电压。包括 W78IE52、W78IE54、W77IC32、W77IE58 等。

1.3.8 Analog Devices 单片机介绍

美国 ADI 公司（Analog Device Inc）公司生产各种高性能的模拟器件，其推出的 8051 内

51 单片机开发与应用技术详解

核的 ADuC800 系列单片机集成了多种精密模拟资源，包括多通道的高分辨率模数转换器 ADC 和数模转换器 DAC、基准电压源和温度传感器等。

ADuC800 系列单片机具有符合工业标准的 8052 MCU 内核，包括 ADuC812、ADuC814、ADuC816、ADuC824、ADuC831、ADuC832、ADuC834、ADuC836、ADuC841、ADuC842、ADuC843、ADuC845、ADuC847、ADuC848 等。

1.3.9 TI 单片机介绍

美国德州仪器（TI）提供两类具有嵌入式 8051/8052 微控制器的产品系列。其中 MicroSystems（MSC）产品系列包括嵌入式数据获取解决方案；TUSB 产品系列包括 USB 嵌入式连接解决方案。

- MicroSystems 系列单片机。这类单片机是完全集成混合信号器件。该系列的产品包括整合了以下组件的 8051 CPU：高精度 delta 型 ADC、高精度 DAC、8 通道复用器、烧坏检测、可选缓冲输入、失调 DAC（数模转换器）、可编程增益放大器（PGA）、温度传感器、精密电压参考、闪速程序存储器、闪速数据存储器 and 数据 SRAM。该产品系列的器件都是引脚兼容的，大大简化了器件移植过程。包括 MSC1200、MSC1201、MSC1202、MSC1210、MSC1211、MSC1212、MSC1213、MSC1214 等。
- USB 接口系列单片机。这类微控制器系列使用标准的 805x 微控制器并将各种外围接口集成到一起，以满足各种 USB 外设需求。所有这些产品都遵从 USB 2.0 规范。其中 TUSB3xxx 器件是 USB 全速适配外围设备。TUSB2136 和 TUSB5052 是将 8052 微控制器和全速 USB 集线器集成到一起的组合 USB 设备。TUSB6xxx 产品是 USB 2.0 高速适配设备。

1.3.10 其他单片机介绍

除了上述的几家半导体公司的单片机外，还有很多其他的半导体厂商也提供了多种型号的 51 内核单片机。例如美国的飞思卡尔 Freescale、摩托罗拉 Motorola、Microchip 等，日本的 NEC、日立 Hitachi、瑞萨 Renesas 等。这些厂商的单片机同样具有不错的性能。

另外，近些年国内的半导体厂商异军突起，也提供了很多有特色的单片机。例如上海普芯达电子有限公司的 CW89F 系列单片机。

上海普芯达电子有限公司总部位于上海张江高科技园区。该公司提供多种半导体器件，包括单片机、电源管理器件、系统监管器件、通信接口器件、信号调理器件、功率驱动器件、数字逻辑器件、存储器、专用标准器件和系统级封装芯片等。其推出的单片机型号有如下两类。

- CW89F 系列单片机。这类单片机具有标准的 8051 内核、12MHz 主频、大电流 I/O、支持 ISP 和四级加密技术。其同时提供了 VML 虚拟固件库是该产品的一个亮点。其将常用的数字模块、模拟模块、通信接口模块等集成在一起，方便了用户的使用。
- CW89FE 系列单片机。这类单片机具有 6T8051 内核、33MHz 主频、大电流 I/O、支持 ISP 和四级加密技术。其同样支持 VML 虚拟固件库，用于减少客户的程序代码，加速程序开发。

这里所介绍的国内外众多单片机厂商推出的主流单片机，都具有很多兼容的特性，但各自有各自的特点，用户可以根据需要选择。尽管单片机家族如此庞大，其实只要熟练掌握一种单片机的使用，便可以举一反三，对其他型号的单片机也能够快速上手。

1.4 51 系列单片机开发概述

单片机应用系统的开发是以单片机为核心，配合一定的外部电路及程序，从而实现特定测量

及控制功能的应用系统。其中单片机的选型、资源分配及程序设计是整个系统设计的关键。一般来说，一个完整的单片机应用系统设计包括分析测控系统、单片机选型、硬件资源分配、单片机程序设计、仿真测试并最终下载到实际硬件电路中执行。单片机开发的整个流程如图 1-1 所示。

1.4.1 分析测控系统

用户在进行单片机应用系统开发时，首先要对该测控系统进行可行性分析及系统总体方案设计。

1. 可行性分析

可行性分析主要是分析整个设计任务的可能性。一般来说，可以通过两种途径进行可行性分析。首先，调研该单片机应用系统或类似设计是否有人做过。如果能找到类似的参考设计，便可以分析其设计思路，并借鉴其主要的硬件及软件设计方案。这样可以在很大程度上减少工作量及自己摸索的时间。如果没有，则需要自己进行整个应用系统的设计；然后，根据现有的硬件及软件条件、自己所掌握的知识等来决定该单片机应用系统是否可行。

2. 系统总体方案设计

当完成可行性分析并确认方案可行后，便进入系统整体方案设计阶段。这里，主要结合国内外相关产品的技术参数和功能特性、本系统的应用要求及现有条件，来决定本设计所要实现的功能和技术指标。接着，制定合理的计划，编写设计任务书，从而完成该单片机应用系统的总体方案设计。

1.4.2 单片机选型

在 51 系列单片机应用系统开发过程中，单片机是整个设计的核心，因此选择合适的单片机型号很重要。目前，市场上的单片机种类很多，不同厂商均推出很多不同侧重功能的单片机类型。在进行正式的单片机应用系统开发之前，需要了解各个不同单片机的特性，从中做出合理的选择。在单片机选型时，主要需要注意以下几点。

- 根据应用系统的硬件资源要求，在性能指标满足的情况下，尽量选择硬件资源集成在单片机内的型号，例如 ADC、DAC、I2C 及 SPI 等。这样便于整个系统的软件管理，可以减少外部硬件的投入，缩小电路板的面积，从而减少投资等。
- 仔细调查市场，尽量选用广泛应用、货源充足的单片机型号，避免使用过时且缺货的型号，这样可以使得硬件投资不会过时。
- 对于手持式设备或其他需要低功耗的设备，尽量选择低电压、低功耗的单片机型号。
- 在条件允许的情况下，尽量选择功能强的单片机，这样便于以后的升级扩展。
- 对于商业性的最终产品，尽量选择体积小的贴片封装的单片机型号，这样可以减少电路板面积，从而降低硬件成本。

1.4.3 硬件资源分配

当总体方案及单片机型号确定下来后，需要仔细规划整个硬件电路的资源分配。一般来说，一个单片机应用系统由紧密联系的硬件及软件构成。因此，在进行设计前，需要规划哪些部分的功能用硬件来实现及用什么硬件来实现，以及哪些部分的功能用软件来实现等。这里需要注意以下几点。

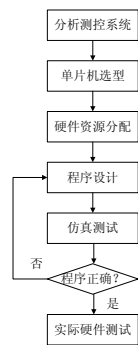


图 1-1 单片机开发流程

51 单片机开发与应用技术详解

- 如果单片机的硬件资源丰富，尽量选择使用单片机内部集成的硬件资源来实现，这样可以减少硬件投资，提高集成度。
- 对于一些常用的功能部件，尽量选择标准化、模块化的典型电路，这样可以提高设计的灵活性，确保成功率等。
- 合理规划单片机的硬件及软件资源，充分发挥单片机的最大功能。
- 硬件上最好留有扩展的接口，以方便后期的维护及升级。
- 要仔细考虑各部分硬件的功率消耗及驱动能力，驱动能力不够将导致系统无法正确运行。

1.4.4 程序设计

在整个单片机应用系统的总体方案及硬件分配定型后，便可以着手进入具体的设计阶段。这里，单片机的程序设计是关键，可以根据实际的需要来选择单片机设计语言及开发环境。在单片机程序设计和时，主要需要从以下几点来考虑。

- 采用结构化的程序设计，将各个功能部件模块化，用子程序来实现，这样便于调试及后续的移植修改等。
- 合理使用单片机的资源，包括 RAM、ROM、定时器/计数器、中断等。
- 尽量采用执行速度快的指令，以充分发挥单片机的性能优势。
- 充分考虑软件运行时的状态，避免未处理的运行状态，否则程序运行时易出错，不受控制。
- 合理安排各个功能部件的时序，确保程序能正确执行。
- 尽量选择使用单片机 C 语言来进行设计，避免使用汇编语言，除非有特殊要求。这样可以使程序易懂，便于交流和后期维护。
- 程序中要尽量添加注释，提高程序的可读性。

1.4.5 仿真测试

单片机仿真测试和程序设计是紧密相关的。在实际设计过程中，需要经常对各个功能部件进行仿真测试，这样可以及时发现问题，确保模块的正确性。对于整个系统的设计，仿真测试则可以模拟实际的程序运行，观察整个时序及运行状态是否合理。当发现问题时，需要返回程序设计阶段修改设计，进而重新仿真测试，直到程序运行通过。

用户在进行程序设计时，需要选择一个好的编译仿真环境，例如 Keil 公司的 μ Vision 系列、英国 Labcenter electronics 公司的 PROTEUS 等。如果条件允许，可以选择一款和单片机型号匹配的硬件仿真器。硬件仿真器一般支持在线仿真调试，可以实时观察程序中的各个变量，对程序进行最大程度的测试。

1.4.6 实际硬件测试

当程序设计通过后，便可以将其下载到单片机中结合整个硬件电路来测试。在实际硬件电路测试阶段，主要看单片机程序和外部硬件接口是否正常，单片机的驱动能力是否够用，以及整个硬件电路的逻辑时序配合是否正确等。如果发现问题，则要返回设计阶段，逐个解决问题。硬件测试通过后，便可以投入使用或生产。

1.5 小结

本章主要介绍了单片机的发展历程、51 系列单片机的简介和 51 系列单片机的应用领域；接着介绍了一下实际常用的一些主流半导体厂商的单片机型号，以供读者参考；最后，详细介绍了开发 51 系列单片机应用系统的整个流程及主要注意事项。

第2章 51 系列单片机基本结构

8051 是 MCS-51 系列单片机中的典型产品，本书将以这一最具代表性的机型进行系统地讲解。其他单片机与 8051 的系统结构相同，只是对 8051 进行一些扩充，使其功能更强、市场竞争力更强。

2.1 51 系列单片机的内部结构

8051 系列单片机的内部结构是各种逻辑单元及其之间的互连构成的。其主要由中央处理器（CPU）、程序存储器（ROM）、数据存储器（RAM）、串行接口、并行 I/O 接口、定时/计数器、中断系统等几大单元，以及数据总线、地址总线和控制总线组成。8051 系列单片机的内部结构框架示意图，如图 2-1 所示。

2.1.1 内部结构的主要组成部分

单片机的性能都反映在单片机所特有的结构和资源上。这里首先介绍一下 8051 单片机基本结构的主要组成部分。

1. 中央处理器（CPU）

中央处理器（CPU）是整个单片机的核心部件。51 系列单片机是 8 位数据宽度的处理器，它能处理 8 位二进制数据或代码。CPU 主要由算术逻辑部件、控制器和专用寄存器三部分电路组成，这将在后面详细介绍。它负责控制、指挥和调度整个单元系统协调的工作，完成运算和控制输入输出功能等操作。

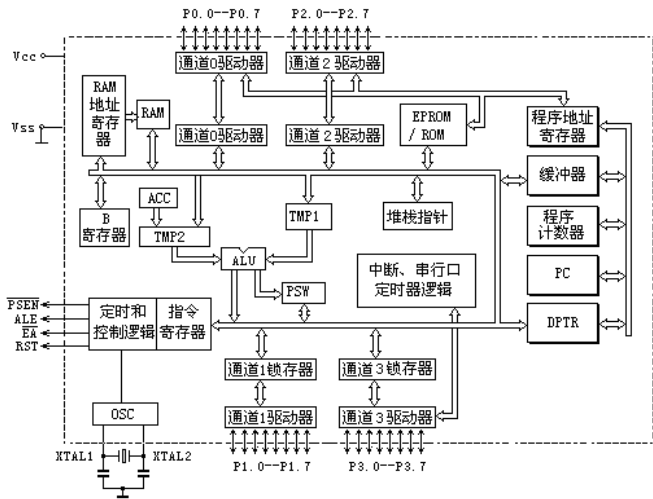


图 2-1 8051 系列单片机的内部结构框图

51 单片机开发与应用技术详解

2. 程序存储器（ROM）

程序存储器（ROM）用于存放用户程序、原始数据或表格等。8051 单片机共有 4096 个 8 位掩膜 ROM，有些增强型的单片机则提供了更大的程序存储器。对于目前新推出的一些单片机，如 AT89S52，其采用的是 Flash 程序存储器。

3. 数据存储器（RAM）

数据存储器（RAM）可存放读写的数据、运算的中间结果或用户定义的字型表等。8051 单片机内部有 128 个 8 位用户数据存储单元和 128 个专用寄存器单元，它们是统一编址的。专用寄存器只能用于存放控制指令数据，用户只能访问，而不能用于存放用户数据。所以，用户能使用的 RAM 只有 128 个。对于一些新型的单片机，内部的 RAM 单元可能更多，例如 AT89S52 内部有 256 个 RAM 数据存储单元。

4. 定时器/计数器

定时器/计数器用于硬件的定时或计数。8051 有两个 16 位的可编程定时/计数器，以实现定时或计数功能。其也可以产生中断，从而在程序中控制程序转向。

5. 并行输入输出（I/O）口

单片机的并行输入输出（I/O）口主要用于和外部设备进行通信，以便于处理外部的输入和将运算结果反馈到外部设备。8051 共有 4 组 8 位 I/O 口（P0、P1、P2 和 P3），用于对数据的读入和输出。

6. 全双工串行口

全双工串行通信口主要用于与其他设备间的串行数据传送。8051 内置一个全双工串行通信口，该串行口既可以用做异步通信收发器，也可以当同步移位器使用。

7. 中断系统

8051 具备较完善的中断功能，有两个外中断、两个定时/计数器中断和一个串行中断。这些功能可满足不同的控制要求，并具有 2 级的优先级别供选择。

8. 时钟电路

8051 内置最高频率达 12MHz 的时钟电路，可以外置振荡晶振和电容，便可以产生整个单片机运行的脉冲时序。设计人员也可以采用外部的时钟源作为工作时钟。

2.1.2 单片机结构的类型

单片机结构的类型可以按照程序存储器和数据存储器的体系结构来划分，大致有如下两种类型。

- 哈佛（Harvard）结构。这种结构程序存储器（ROM）和数据存储器（RAM）采用分开形式。哈佛结构便于对程序和数据同时访问，可以减少程序运行时的访问瓶颈，提高数据的吞吐率。
- 普林斯顿（Princeton）结构。这种结构中采用通用计算机广泛使用的程序存储器（ROM）与数据存储器（RAM）合二为一的方式，也称为冯 诺伊曼结构。由于程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，因此程序指令和数据的宽度相同。

例如，Intel 的 MCS-51 系列单片机普遍采用的是哈佛结构，而后续产品 16 位的 MCS-96 系列单片机则采用普林斯顿结构。

2.2 51 系列单片机的引脚功能

Intel 的 8051 是一个典型的单片机，在后续 MCS-51 系列单片机的产品中，均以其为核心电路发展起来的。它们具有和 8051 一致的硬件结构和软件特征。

随着半导体技术的发展，各种新型的单片机层出不穷，性能也得到不断提升，价格也越来越低。因此，本章在介绍 51 系列单片机时不局限于介绍基本的 8051，而是以最新型的 51 系列单片机来进行介绍。这些新型的单片机在基本功能上和 8051 完全一致，从而使读者在理解 8051 结构的同时，也熟悉了最新的单片机。

2.2.1 51 系列单片机的引脚封装

MCS-51 系列单片机有各种封装形式，这里均以双列直插 DIP 形式的封装来进行介绍。Intel 的 8051 的典型引脚配置，如图 2-2 所示。这是 40Pin 封装的双列直插 DIP 式结构，其中正电源和地线两根，外置石英振荡器的时钟线两根，4 组 8 位共 32 个 I/O 口，中断口线与 P3 口线复用。

Atmel 公司的 AT89S 系列与 MCS-51 系列是兼容的，这里主要以 AT89S52 单片机来介绍 51 系列单片机的基本原理。AT89S52 的 40Pin 双列直插封装的引脚配置，如图 2-3 所示。

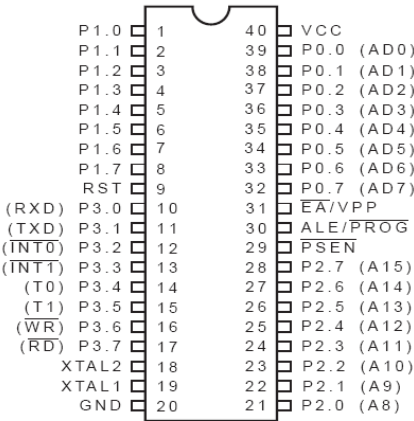


图 2-2 8051 双列直插式的引脚配置

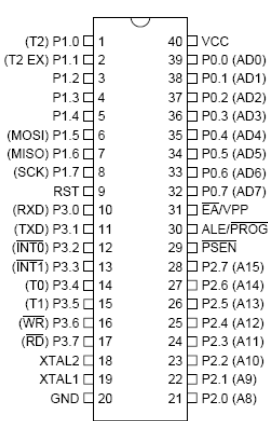


图 2-3 AT89S52 的双列直插式的引脚配置

从图 2-2 和图 2-3 中可以看出，AT89S52 和 8051 各个引脚的基本功能是完全一致的，不同的是 AT89S52 内部程序存储器和数据存储器的空间更大，而且部分引脚有了扩展功能，这在下面的章节中会详细介绍。

2.2.2 51 系列单片机引脚功能

这里以 AT89S52 为原型，介绍 51 系列单片机的引脚功能。虽然市场上 51 系列单片机种类很多，但由于 51 系列单片机的引脚都基本兼容，因此这些内容同样适合于其他型号的单片机。

1. 电源引脚

电源引脚主要负责单片机的供电，有两根引脚。

- VCC (Pin40)：电源端。正电源接 4.0~5.0V 电压，正常工作电压为+5V。
- GND (Pin20)：接地端。

2. 外接晶振或外部振荡器引脚

外接晶振或外部振荡器引脚主要负责为单片机的运行提供时钟振荡器，主要有两根引脚。

➤ XTAL1 (Pin19)：时钟 XTAL1 脚，片内振荡电路的输入端。

➤ XTAL2 (Pin18)：时钟 XTAL2 脚，片内振荡电路的输出端。

AT89S52 的时钟振荡器有两种工作方式。一种是片内时钟振荡方式，在 18 和 19 脚外接石英晶体（0~33MHz）和振荡电容，振荡电容的值一般取 10pf~30pf。另外一种方式是外部时钟方式，将 XTAL2 悬空，外部时钟信号（0~33MHz）从 XTAL1 脚输入。

3. 输入输出（I/O）端口引脚

AT89S52 提供了 4 个 8 位并行 I/O 端口，基本的功能和其他 51 系列单片机完全一致，部分引脚有扩展功能。分别介绍如下：

➤ P0 口：即 P0.0~P0.7 (Pin39~Pin32)，输入输出脚，可用于 8 位并行 I/O 口或分时复用为地址和数据总线。P0 口作为输出口时，每个引脚负载 8 个 TTL；在外扩存储器时，可定义为低 8 位地址/数据线；定义为 I/O 口时，需外接上拉电阻，为准双向 I/O 口，在程序中向该端口写入“1”后，成为高阻抗输入口；在对片内 Flash 编程时，P0 口可以接收字节代码，在程序校验时输出字节代码，程序校验期间应外接上拉电阻。

➤ P1 口：即 P1.0~P1.7 (Pin1~Pin8)，输入输出脚，8 位准双向并行 I/O 口。P1 口内部已经具有上拉电阻的 8 位准双向 I/O 口，能负载 4 个 TTL；在 Flash 编程和校验时，定义为低 8 位地址线。和基本的 8051 不同的是，其部分引脚具有第二功能。引脚 P1.0 是定时/计数器 T2 的外部计数输入，引脚 P1.1 表示定时/计数器 T2 捕获/再装入触发及方向控制，引脚 P1.5 (MOSI)、P1.6 (MISO) 和 P1.7 (SCK) 用于系统内部编程。

➤ P2 口：即 P2.0~P2.7 (Pin21~Pin28)，输入输出脚，8 位准双向并行 I/O 口。P2 口内部已经具有上拉电阻的 8 位准双向 I/O 口，能负载 4 个 TTL；当访问外部存储器时，定义为高 8 位地址线。如果只需 8 位地址线，它将输出特殊功能寄存器（锁存器）中的内容。

➤ P3 口：即 P3.0~P3.7 (Pin10~Pin17)，输入输出脚，8 位准双向并行 I/O 口。P3 口内部已经具有上拉电阻的 8 位准双向 I/O 口，能负载 4 个 TTL。和普通的 8051 一样，每个引脚都具有第二功能。引脚 P3.0 (RXD) 和引脚 P3.1 (TXD) 用于串行数据传输，分别为串行数据的接收和发送端口；引脚 P3.2 和引脚 P3.3 为外部中断请求，分别用于 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ 的中断输入；引脚 P3.4 (T0) 和引脚 P3.5 (T1)，分别为定时器/计数器 0 和定时器/计数器 1 的外部计数输入端；引脚 P3.6 ($\overline{\text{WR}}$) 和引脚 P3.7 ($\overline{\text{RD}}$) 用于读写单片机外部 RAM，分别是外部数据写选通信号和读选通信号。

4. 复位、控制和选通引脚

复位、控制和选通引脚主要负责单片机程序复位、编程控制及外部程序存储器的选通。下面分别进行介绍。

➤ RST (Pin9)：单片机内部 CPU 的复位信号输入端。

在单片机的振荡器启动后，该引脚置两个机器周期以上高电平，便可以实现复位。另外，对于 AT89S52，其内部包含定时监视器（看门狗）电路。在定时监视器定时输出后，该引脚置高电平，并持续 96 个振荡周期，也可以实现复位。

特殊功能寄存器 AUXR 中的 DISRT0 位可以使复位无效。在默认的 DISRT0 位状态下，RST 引脚上的高电平有效。

➤ ALE/ $\overline{\text{PROG}}$ (Pin30)：ALE 为地址锁存使能端和编程脉冲输入端。

- 当访问外部程序存储器时，ALE（地址锁存）的负跳变将低 8 位地址打入锁存；而非访问内部程序存储器时，ALE 端将有一个 1/6 振荡频率的正脉冲信号，该信号可以用于外部计数或时钟信号。当访问外部数据存储器（执行 MOVX 类指令）时，ALE 会跳过一个脉冲。

在 Flash 编程时，该引脚用于输入编程脉冲（ $\overline{\text{PROG}}$ ）。

另外，在程序中，可对特殊功能寄存器区的地址为 8EH 单元的 D0 位置“1”，可禁止 ALE 输出，只有在执行 MOVX 或 MOVC 类指令时，ALE 才被激活，仍输出锁存有效，在执行外部程序时，该设定禁止 ALE 位无效。

- $\overline{\text{PSEN}}$ (Pin29)：访问外部程序存储器读选通信号。

当单片机访问外部程序存储器，读取指令码时，每个机器周期产生两次有效信号，即此脚输出两个负脉冲选通信号；在执行片内程序存储器读取指令码时，不产生此脉冲；在读写外部数据时，也不产生 $\overline{\text{PSEN}}$ 脉冲信号。

- $\overline{\text{EA/Vpp}}$ (Pin31)： $\overline{\text{EA}}$ 为访问内部或外部程序存储器选择信号，在 Flash 编程时，提供 Flash 编程电压 V_{pp} 。

当 CPU 访问外部程序存储器 (0000H~FFFFH 单元) 时，则 $\overline{\text{EA}}$ 必须保持低电平（即接 GND 端）；当 $\overline{\text{EA}}$ 保持高电平（即接 VCC 端）时，则 CPU 先从片内 0000H 单元开始，执行内部程序存储器程序；如果外部还有扩展程序存储器，则 CPU 在执行完内部程序存储器程序后，自动转向执行外部程序存储器程序；对片内 Flash 编程时，此引脚用于输入编程允许电压。另外，如果单片机的保密位被编程，则复位时，内部会锁存 $\overline{\text{EA}}$ 端的状态。

2.3 中央处理器 (CPU)

中央处理器 (CPU) 是由算术逻辑部件 (ALU)、控制器和寄存器通过总线连接而成的一个整体。中央处理器是整个单片机的核心部件。CPU 负责控制、指挥和调度整个单元系统协调的工作，完成运算和控制输入输出功能等操作。

51 系列单片机是 8 位数据总线的处理器，能处理 8 位二进制数据或代码。AT89S52 的 CPU 与 8051 的 CPU 完全兼容，这里以 AT89S52 为例进行介绍。

2.3.1 算术逻辑部件 (ALU)

算术逻辑部件是进行算术及逻辑运算的功能部件。AT89S52 单片机的算术逻辑部件包括运算器、累加器 A、寄存器 B、暂存器 TMP、程序状态寄存器 PSW、堆栈指针 SP、数据指针 DPTR 等。其可以进行加、减、乘、除四则运算，也可以进行与、或、非、异或等逻辑运算，还可以执行数据传送、移位、判断和程序转移等功能。

51 系列单片机的 ALU 提供了丰富的指令系统和极快的指令执行速度，大部分指令执行时间为 1 μ s，乘法指令为 4 μ s。AT89S52 的 ALU 与 8051 的 ALU 完全兼容，其位处理功能非常强，这对“面向控制”特别有用，指令功能极为丰富，8 位并行处理能力极强。

2.3.2 控制器

控制器是用来统一指挥和控制计算机进行工作的部件。其主要的功能是从程序存储器中提取指令，送到指令寄存器，再送入指令译码器进行译码。控制器通过定时和控制电路，在规定的时刻发出各种操作所需要的全部内部控制信息及 CPU 外部所需要的控制信号，如 ALE、 $\overline{\text{PSEN}}$ 、RD、WR 等，使各部分协调工作，完成指令所规定的各种操作。

8051 的控制器包括时钟发生器、定时控制逻辑、指令寄存器、指令译码器、程序计数器 PC、程序地址寄存器、数据指针寄存器 DPTR 和堆栈指针 SP 等。

2.3.3 通用寄存器

寄存器是用来存放信息的单元，其优点是存取速度快、方便，寄存器的数量是衡量一台计算机处理能力的重要标志。51 系列单片机的寄存器可分为工作寄存器（即通用寄存器）、专用寄存器和特殊功能寄存器（SFR）。特殊功能寄存器是专为对应功能服务，在第 2.4.2 节“数据存储及其扩展”一节将详细介绍。本节主要介绍通用寄存器 R0~R7。

片内 RAM 的 00H~1FH（共 32B）空间，开辟了 4 个通用寄存器区，每组共有 8 个 8 位的寄存器 R0~R7，通过对 PSW 的 RS1 和 RS2 进行设置，选择任意一组使用，同时其余三组被屏蔽。其优点是，避免进栈保护、减少堆栈深度、节省出入栈指令和时间。不用的通用寄存器可作为片内 RAM 访问。

2.3.4 专用寄存器

专用寄存器是专门为某些功能部件设计的寄存器。其主要包括程序计数器 PC、累加器 A、寄存器 B、程序状态寄存器 PSW、堆栈指针 SP、数据指针 DPTR 等，下面分别介绍这几种寄存器的功能。

1. 程序计数器 PC

程序计数器 PC 是一个 16 位二进制的程序地址寄存器，专门用来存放下一条需要执行指令的内存地址，能自动加 1。

CPU 执行指令时，根据程序计数器 PC 中的地址从存储器中取出当前执行的指令码，将其送给控制器分析执行，随后程序计数器 PC 中的地址码自动加 1，以便取下一条指令码。这样一次次加 1，指令就被一条条执行。

8051 的程序计数器 PC 是由 16 个触发器构成，其编码范围为 0000H~FFFFH，其寻址范围为 64KB。

2. 累加器 A

累加器 A（或者 ACC）是运算过程中的暂存寄存器，是一个 8 位二进制寄存器，用于提供操作数和存放操作结果。通过内部总线直接与 ALU 相连，一般的信息传递和交换均需通过累加器 A。

3. 寄存器 B

寄存器 B 一般用于乘除法操作指令，也是一个 8 位二进制寄存器，由 8 个触发器组成，与累加器 A 配合使用。寄存器中存放乘数或除数、乘积的高位字节和除法的余数。其他情况下，可作为一般寄存器或中间结果暂存器使用。

4. 程序状态寄存器 PSW

程序状态寄存器 PSW 是一个 8 位寄存器，用于存放指令执行后的有关状态，为后面的指令执行提供状态条件。PSW 中的各状态位通常是在指令执行过程中自动形成的，用户可以根据需要改变 PSW 的状态。PSW 中的各状态标志位如图 2-4 所示。

PSW 中的各状态标志位说明如下：

- Cy：高位进位标志位。如果指令运算结果高位产生进位或借位，则 Cy 被置为“1”，否则为“0”。它在位处理指令中作为累加器用，所以又称为“布尔累加器”。
- AC：辅助进位标志位。根据指令运算结果的低 4 位即 D3 有无向高 4 位 D4 进位或借位，

来进行置位。如果有进位或借位，则置 AC 为“1”，否则为“0”。AC 也常称之为半字节进位标志位，常用于 BCD 码调整。

- F₀：用户标志位。该标志位不是在指令执行过程中自动形成的，是由用户根据需要通过传送指令设置。该标志位可用于决定用户程序的走向。
- RS₁ 和 RS₀：工作寄存器选择标志位。用于选择内部 RAM 的 4 组工作寄存器的某组置于前台，每个寄存器由 8 个 8 位寄存器（R0~R7）组成。通过改变 RS₁ 和 RS₀，可以决定通用寄存器的实际物理地址。这两者之间的关系如表 2-1 所示。



图 2-4 PSW 中的状态标志位

表 2-1 RS₁、RS₀ 与寄存器 R0~R7 间对应关系

RS ₁	RS ₀	寄存器组	R0~R7 的物理地址
0	0	0 组	00H~07H
0	1	1 组	08H~0FH
1	0	2 组	10H~17H
1	1	3 组	18H~1FH

- OV：溢出标志位。当运算结果数值的绝对值超过允许的最大值时，就会产生溢出。由机器在程序执行过程中自动形成。当运算结果数的 A7 或 A6 其中的一位产生进位或借位时，OV 位自动置“1”，否则置“0”。其逻辑表达式为 $OV=A_7\oplus A_6$ 。
- —：保留位，无定义，用户不能使用。
- P：奇偶校验标志位。逻辑运算后，运算结果中“1”的个数为奇数，则 P 位自动置“1”，否则置“0”。

5. 堆栈指针 SP

堆栈指针 SP 是在片内 RAM 中开辟一个存储区域，专门存放堆栈栈顶的地址。其中堆栈是一组编有地址码的特殊存储单元。堆栈指针 SP 采用 8 位增量的寄存器，堆栈深度为 0~255 个存储单元。

数据进栈时，SP 自动增 1，将数据压入 SP 所指向的堆栈单元；弹出时，将 SP 所指向的堆栈单元内的数据推出栈，然后 SP 自动减 1。SP 总是指向栈顶。堆栈按照“先进后出、后进先出”的原则进行数据存取。

在系统复位后，堆栈指针 SP 的初始值为 07H，即栈底为 08H 单元。这样就与工作寄存器区域重叠，必须重新定义 SP，在片内 RAM 中开辟一个合适的堆栈区域。

6. 数据指针 DPTR

数据指针 DPTR 是一个 16 位寄存器，是由 8 位寄存器 DPH 和 DPL 组合而成。其中 DPH 为 DPTR 的高 8 位，DPL 为 DPTR 的低 8 位。它既可以作为 16 位数据指针用，也可以分开以 8 位寄存器（DPH、DPL）单独使用。DPTR 可以用来存放片内 ROM 的地址，也可以存放片外 RAM 和片外 ROM 的地址。

AT89S52 设有两个 DPTR，分别为 DP0（DP0H、DP0L）和 DP1（DP1H、DP1L），通过对特殊功能寄存器 AUXR₁ 的 DPS 位进行设置选择。当 DPS=0，选择 DP0；当 DPS=1，选择 DP1。

2.4 存储器结构

存储器是单片机的三大主要部件之一，主要用来储存信息（即数据和程序）。存储器按配置方法分为主存和外存。“主存”又称为“内存”，用于存放当前执行的数据和程序；“外存”用于存放暂不执行的数据和程序。目前单片机主要配置内存。

存储器结构分为独立的两部分，即数据存储器（RAM）和程序存储器（ROM）。8051 单片机的存储器可分为 4 个存储空间，即片内程序存储器（片内 ROM）、片外程序存储器（片外 ROM）、片内数据存储器（片内 RAM）、片外数据存储器（片外 RAM）。

这 4 类存储器与其对应的地址关系，如表 2-2 所示。

表 2-2 RS₁、RS₀ 与寄存器 R0~R7 间对应关系

存 储 器	物理地址
4KB 片内程序存储器（片内 ROM）	0000H~0FFFFH
64KB 片外程序存储器（片外 ROM）	0000H~FFFFH（其中 1000H~FFFFH 为外部 ROM）
256B 片内数据存储器（片内 RAM）	00H~FFH
64KB 片外数据存储器（片外 RAM）	0000H~FFFFH

8051 单片机片内有 4KB 的程序存储器和 256B 的数据存储器，还可以片外扩展至 64KB 程序存储器和 64KB 数据存储器。关于这部分内容将在系统扩展部分介绍。

2.4.1 程序存储器及其扩展

程序设计人员编写的程序代码就存放在单片机的程序存储器中，也称为“只读程序存储器”（ROM）。程序和数据一样，都是由机器码组成的代码串，只是程序代码存放于程序存储器中。

1. 程序存储器

51 系列单片机具有 64KB 程序存储器寻址空间，这 64KB 的地址空间是统一编址的，没有采用片内、片外分区的方式。区分片内、片外是由 EA 引脚上的电平来指示，具体如下：

- EA=1，即接高电平时，CPU 从片内的程序存储器中读取程序，当 PC 值超过片内 ROM 的容量时，才会转向外部的程序存储器读取程序。
- EA=0，即接低电平时，CPU 从片外的程序存储器中读取程序，并输出 PSEN 选通信号。对于内部无 ROM 的 8031 单片机，其 ROM 只能外接，必须使 EA=0。
- 程序存储器是由 16 位的程序计数器 PC 指示当前地址。片内 ROM 的地址为 0000H~0FFFFH。单片机启动复位后，程序计数器 PC 的内容为 0000H，系统将从 0000H 单元开始执行程序。

在程序存储器中的 0003H~0032H，共 48B 被保留专用于中断处理程序，称为中断矢量区，系统必须跳过这一区域。其中，

- 0000H~0002H 单元：系统复位后，PC 为 0000H，单片机从 0000H 单元开始执行程序，如果程序不是从 0000H 单元开始执行，则应在这三个单元中存放一条无条件转移指令，让系统必须跳过这一区域，直接去执行用户指定的程序。
- 0003H~002AH：这 40 个单元各有用途，被均匀地分为 6 段，其定义如表 2-3 所示。

表 2-3 中断入口地址

地 址	用 途
0003H	外部中断 0（INT0）中断入口地址

000BH	定时/计数器 0 (TF ₀) 中断入口地址
-------	------------------------------------

续表

地 址	用 途
0013H	外部中断 1 ($\overline{\text{INT1}}$) 中断入口地址
001BH	定时/计数器 1 (TF ₁) 中断入口地址
0023H	串行中断 (T ₁ 、R ₁) 入口地址
002BH	定时/计数器 2 (TF ₂ 、EXF ₂) 中断入口地址, 在 AT89S52 中使用

以上地址单元被专门用于存放中断处理程序。中断响应后, 按中断的类型自动转到各自的中断区去执行程序。这些地址单元不能用于存放程序的其他内容, 只能存放中断服务程序。通常情况下, 每段只有 8 个地址单元是不能保存完整的中断服务程序, 因而一般在中断响应的地址区, 存放一条无条件转移指令, 指向程序存储器中真正存放中断服务程序的空间。这样中断响应后, CPU 读到这条转移指令, 便转向真正存放中断服务程序的空间, 继续执行中断服务程序。

2. 程序存储器的扩展

51 系列单片机为了满足不同应用的需要, 除了设置有内部程序存储器外, 还可以根据需要进行外部程序存储器扩展。外部程序存储器扩展时, 采用 P0 和 P2 作为 16 位地址总线的低 8 位和高 8 位, 另外, P0 口还分时复用为 8 位数据总线。

与普通的 8051 单片机不同, AT89S52 内部设有 8KB 的可擦写 Flash 存储器。这个程序存储器可以外部扩展至 56KB, 这时需要将单片机的 $\overline{\text{EA}}$ 引脚接高电平。在外部扩展程序存储器情况下, 程序可以首先从片内的程序存储器开始顺序执行, 访问时 CPU 会自动转向外部程序存储器。

当 CPU 访问外部程序存储器的时候, 程序存储器指针 PC 的低 8 位地址由 P0 口输出, PC 的高 8 位地址由 P2 口输出。P2 口和 P1 口共同组成 16 位地址总线。外部程序存储器中的指令代码由 P0 口输入, 即 P0 口是低 8 位地址线和 8 位数据线的分时复用。

为了保证在访问外部程序存储器期间, 16 位的地址码不变, 并且能正确地从 P0 口读入程序代码, 应该将 P0 口输出的低 8 位地址在 ALE 信号的控制下存入地址锁存器, 这样便可以将 P0 口空出来读取 8 位的程序代码。

如果 $\overline{\text{EA}}$ 引脚接低电平, 则 CPU 直接从片外程序存储器的 0000H 地址开始执行, 而不管片内 8KB 的 Flash 是否含有程序代码。

2.4.2 数据存储器及其扩展

数据存储器也称为“随机存取数据存储器”。51 系列单片机的数据存储器在物理逻辑上分为两个地址空间, 即片内数据存储区和片外数据存储区。片内 RAM 有 256B 的用户数据存储区域 (不同的型号有分别), 是用于存放执行的中间结果和过程数据的 51 系列单片机的。数据存储器均可读写, 部分单元还可以位寻址, 其结构示意图如图 2-5 所示。

51 系列单片机内部 RAM 共有 256 个单元 (不同的型号有分别), 这 256 个单元按其功能分为低 128B 片内 RAM 和高 128B 片内 RAM 两部分。

1. 低 128B 片内 RAM

低 128B 片内 RAM: 地址空间为 00H~7FH 单元, 为用户数据 RAM, 可以存放运算结果和标志位等。该区域按

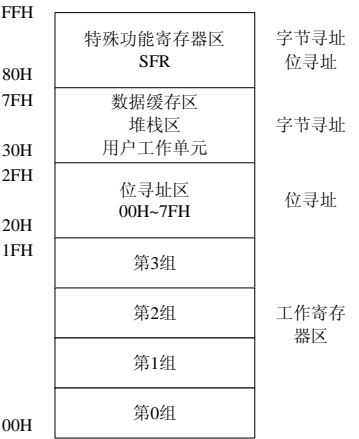


图 2-5 数据存储器结构示意图

51 单片机开发与应用技术详解

其功能还可分为以下三个区域。

- 00H~1FH: 通用寄存器区，开辟了 4 组通用寄存器，每组 R0~R7，在前面寄存器一节中进行过详细介绍。
- 20H~2FH: 位寻址区，共 16B，128 位。布尔处理的存储空间就是位寻址区。既可作为一般单元用字节寻址，也可以进行寻址。该区域除了作为一般 RAM 进行读写外，还可进行执行置“1”、清“0”、求反、转移，传送和逻辑等位操作。位寻址区地址表如表 2-4 所示。

表 2-4 位寻址区位地址表

单元地址	MSB 位地址 LSB							
2FH	7FH	7EH	7DH	7CH	7BH	7AH	79H	78H
2EH	77H	76H	75H	74H	73H	72H	71H	70H
2DH	6FH	6EH	6DH	6CH	6BH	6AH	69H	68H
2CH	67H	66H	65H	64H	63H	62H	61H	60H
2BH	5FH	5EH	5DH	5CH	5BH	5AH	59H	58H
2AH	57H	56H	55H	54H	53H	52H	51H	50H
29H	4FH	4EH	4DH	4CH	4BH	4AH	49H	48H
28H	47H	46H	45H	44H	43H	42H	41H	40H
27H	3FH	3EH	3DH	3CH	3BH	3AH	39H	38H
26H	37H	36H	35H	34H	33H	32H	31H	30H
25H	2FH	2EH	2DH	2CH	2BH	2AH	29H	28H
24H	27H	26H	25H	24H	23H	22H	21H	20H
23H	1FH	1EH	1DH	1CH	1BH	1AH	19H	18H
22H	17H	16H	15H	14H	13H	12H	11H	10H
21H	0FH	0EH	0DH	0CH	0BH	0AH	09H	08H
20H	07H	06H	05H	04H	03H	02H	01H	00H

- 30H~7FH: 字节寻址区，共 80B，用户使用的一般 RAM，可在此区域开辟堆栈。

2. 高 128B 片内 RAM

高 128B 片内 RAM: 地址空间为 80H~FFH 地址单元，为特殊功能寄存器（SFR）区。8051 内部有 21 个特殊寄存器，AT89S52 有 32 个特殊寄存器，如表 2-5 所示。


表 2-5 特殊功能寄存器（SFR）

符 号	特殊功能寄存器名称	地 址
*ACC	累加器	E0H
*B	乘法寄存器	F0H
*PSW	程序状态字	D0H
SP	堆栈指针	81H
DP0L	数据指针 DPTR0 低 8 位	82H
DP1L	数据指针 DPTR1 低 8 位	84H
DP0H	数据指针 DPTR0 高 8 位	83H
DP1H	数据指针 DPTR1 高 8 位	85H
*IE	中断允许控制器	A8H

*IP	中断优先控制器	D8H
-----	---------	-----

续表

符 号	特殊功能寄存器名称	地 址
*P0	I/O 端口 0	80H
*P1	I/O 端口 1	90H
*P2	I/O 端口 2	A0H
*P3	I/O 端口 3	B0H
PCON	电源控制及波特率选择	87H
*SCON	串行口控制器	98H
SBUF	串行数据缓冲器	99H
*TCON	定时/计数器控制	88H
*T2CON	定时/计数器 2 控制	C8H
TMOD	定时/计数器方式选择	89H
T2MOD	定时/计数器 2 方式选择	C9H
TL0	定时/计数器 0 低 8 位	8AH
TL1	定时/计数器 1 低 8 位	8BH
TH0	定时/计数器 0 高 8 位	8CH
TH1	定时/计数器 1 高 8 位	8DH
TL2	定时/计数器 2 低 8 位	CCH
TH2	定时/计数器 2 高 8 位	CDH
RCAP2L	定时/计数器 2 陷阱寄存器低字节	CAH
RCAP2H	定时/计数器 2 陷阱寄存器高字节	CBH
AUXR	辅助寄存器	8EH
AUXR1	辅助寄存器 1	A2H
WDTAST	WDT（看门狗）控制	A6H

说明：带*号的特殊功能寄存器都是可以位寻址的寄存器。

特殊功能寄存器分散在 30H~7FH 片内 RAM 区，其中有部分地址单元未定义，不能使用。访问特殊功能寄存器使用直接寻址方式，其中有一部分（表中带*号）特殊功能寄存器也可以采用位寻址，其特征是地址能被 8 整除。访问这些寄存器中的各位时，在位寻址指令中，可以用“寄存器名.位”、“字节地址.位”、“位地址”、“位名称”等来表示。例如“B.5”表示寄存器 B 的第 5 位。可位寻址的特殊功能寄存器及其位地址如表 2-6 所示。

表 2-6 可位寻址的特殊功能寄存器及其位地址

B	F7H	F6H	F5H	F4H	F3H	F2H	F1H	F0H	F0H
ACC	E7H	E6H	E5H	E4H	E3H	E2H	E1H	E0H	E0H
PSW	D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H	D0H
	Cy	AC	F0	RS1	RS0	OV	—	P	
T2CON	CFH	CEH	CDH	CCH	CBH	CAH	C9H	C8H	C8H
	—	—	PT2	PS	PT1	PX1	PT0	PX0	

IP	BFH	BEH	BDH	BBH	BBH	BAH	B9H	B8H	B8H

续表

P3	B7H	B6H	B5H	B4H	B3H	B2H	B1H	B0H	B0H
	EA	—	ET2	ES	ET1	EX1	ET0	EX0	
IE	AFH	AEH	ADH	AAH	AAH	AAH	A9H	A8H	A8H
P2	A7H	A6H	A5H	A4H	A3H	A2H	A1H	A0H	A0H
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI	
SCON	9FH	9EH	9DH	9CH	9BH	9AH	99H	98H	98H
P1	97H	96H	95H	94H	93H	92H	91H	90H	90H
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	
TCON	8FH	8EH	8DH	8CH	8BH	8AH	88H	88H	88H
P0	87H	86H	85H	84H	83H	82H	81H	80H	80H

3. 外部数据存储器扩展

51 系列单片机为了满足不同应用的需要，除了设置有内部数据存储器外，还可以根据需要
进行外部数据存储器扩展。外部数据存储器扩展时，采用 P0 和 P2 作为 16 位地址总线的低 8
位和高 8 位，另外，P0 口还分时复用为 8 位数据总线。

对于 AT89S52 单片机，如果其内部的 256B 的 RAM 不够用，外部还可以扩展 64KB 的数
据存储器空间。

由于访问外部数据存储器是随机的，因此其访问地址可由工作寄存器 R0 或 R1 来寻址 256B
单元或由数据指针 DPTR 寻址 54KB 的空间。

访问外部数据存储器的硬件结构和访问外部程序程序器相同。R0 或 R1 的 8 位地址由 P0
口送入地址锁存器输出，DPTR 的 16 位地址则由 P0 口输出低 8 位地址到锁存器中，P2 口则输
出高 8 位地址。读写的数据由 P0 口输入输出，因此 P0 口仍然是地址/数据分时复用的。

访问外部数据 MOVX 类指令是单字节双周期指令，因此在第二个机器周期 ALE 信号将减
少一次。

访问外部数据存储器的读写选通信号为 $\overline{RD}/\overline{WR}$ ，均为低电平有效。访问外部程序和数据
存储器各有不同的地址指针 PC、R_i、DPTR，各指针又有不同的读或读写选通信号 \overline{PSEN} 、
 $\overline{RD}/\overline{WR}$ 。这就从结构上把程序存储器和数据存储器的访问截然分开。两者虽然同用 16 位地
址总线，由于使用两套完全不同的地址指针和读、读写选通信号，从而保证了指令的执行不会
出错，为单片机的应用提供了方便。

访问内部 RAM 用 MOV 类指令，是访问片内 RAM 还是特殊功能寄存器则由寻址方式来
区分。

- 访问特殊功能寄存器只能用直接寻址方式；
- 访问外部数据存储器则需选用 MOVX 类指令；
- 对于只需寻址 256B 单元的可采用 R_i 进行 8 位地址间接寻址，否则选用 DPTR16 位地
址指针，寻址 64KB 地址空间；
- 读取程序存储器中的固定数据，则需选用 MOVC 类指令。

可见，访问不同地址空间的数据需要采用不同的指令类型。

2.5 51 系列单片机的复位

复位是使单片机的 CPU 及系统的各个部件处于特定的初始状态，并使系统从初始状态开始工作。单片机的复位是一个很重要的内容，一般在系统上电，或者程序死机的时候需要进行单片机的复位。

2.5.1 单片机的复位状态

单片机的复位状态是单片机在上电时，首先进入的一个特定的状态。在复位状态下，CPU 和整个硬件资源，特别是特殊功能寄存器都处于初始化的状态。如表 2-7 所示列出了在单片机复位状态下的初始值。

表 2-7 单片机的复位状态

特殊功能寄存器	复位状态	特殊功能寄存器	复位状态
ACC	00H	TH0	00H
B	00H	TL0	00H
DPTR	0000H	TH1	00H
PC	0000H	TL1	00H
PSW	00H	TMOD	00H
P0~P3	FFH	TCON	00H
SP	07H	SCON	00H
IE	0XX0 0000B	PCON	0XXX 0000B
IP	XXX0 0000B	SBUF	XXXX XXXXB
T2MOD*	XXXX XX00B	TH2*	00H
T2CON*	00H	TL2*	00H
RCAP2L*	00H	AUXR*	XXX0 0XX0B
RCAP2H*	00H	AUXR1*	XXXX XXX0B
WDTRST*	XXXX XXXXB		

其中，带“*”号的为 AT89S52 所特有的，其余适用于所有的 8051。从表 2-7 中可以看出，在复位状态下，PC=0000H，表示单片机 CPU 将从 0000H 单元开始向下执行程序。

2.5.2 单片机的复位电路

单片机的复位电路是促使单片机进入复位状态的硬件结构。单片机的复位是很重要的，复位操作可以完成单片机的初始化，也可使处于死机状态下的单片机重新开始运行。

1. 复位要求

单片机复位的原理是在时钟电路开始工作后，在单片机的 RST 引脚施加 24 个时钟振荡脉冲（即两个机器周期）以上的高电平，单片机便可以实现复位。在复位期间，单片机的 ALE 引脚和 PSEN 引脚均输出高电平。当 RST 引脚从高电平跳变为低电平后，单片机便从 0000H 单元开始执行程序。

在实际应用中，一般采用外部复位电路来进行单片机复位。一般在 RST 引脚保持 10ms 以上的高电平，保证单片机能够可靠地复位。单片机的复位电路可以有上电复位、手动加上电复位、看门狗复位及一些复杂的复位电路。下面分别进行介绍。

2. 上电复位电路

上电复位电路的基本原理是利用 RC 电路的充放电效应，电路如图 2-6 所示。当单片机系统上电的时候，复位电路通过电容加在 RST 引脚一个短暂的高电平信号，这个高电平信号随着电容的充电而逐渐降低，这个高电平持续的时间和 RC 电路的充放电时间有关。

3. 手动加上电复位电路

在实际应用的电路中，一般采用既可以手动复位，又可以上电复位的电路，这样可以人工复位单片机系统。这种电路如图 2-7 所示。上电复位部分的原理也是 RC 电路的充放电效应。除了系统上电的时候可以给 RST 引脚一个短暂的高电平信号外，当按下按键开关的时候，VCC 通过一个电阻连接到 RST 引脚，给 RST 一个高电平，按键松开的时候，RST 引脚恢复为低电平，复位完成。

4. 定时监视器（WDT，看门狗）复位

定时监视器复位是采用单片机内部的看门狗来实现的复位操作。近年来新出的新型单片机均包含看门狗 WDT，WDT 可以根据应用程序的运行周期来设定。当应用程序在运行过程中，由于外界的干扰而进入非正常工作状态时，WDT 定时计数器产生溢出信号，复位单片机，重新恢复正常运行。

对于自身不带看门狗功能的单片机，可以采用专门的复位电路芯片，如 MAXIM 公司的 MAX813L。MAX813L 是带有看门狗和电源监控功能的复位芯片，具体的用法可以参考该芯片的资料手册。

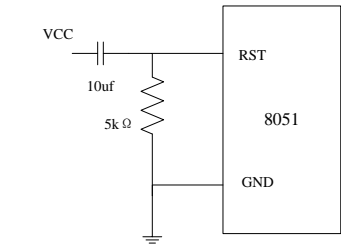


图 2-6 上电复位电路

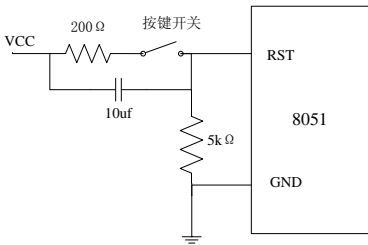


图 2-7 手动加上电复位电路

5. 复杂的复位电路

对于前面介绍的简单的复位电路，干扰很容易传入复位端。虽然在大多数情况下不会引起单片机错误复位，但有时会使某些寄存器错误复位。因此，在一些要求严格的场合，需要对单片机的复位电路进行仔细设计，或者采用专用的复位芯片来完成，如图 2-8 和图 2-9 所示的便是两个例子。

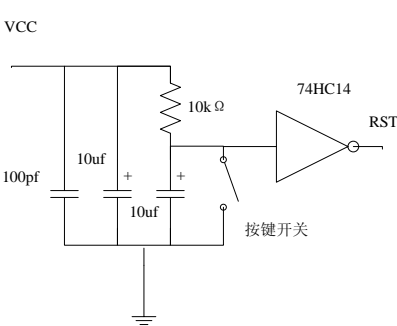


图 2-8 复杂的复位电路 1

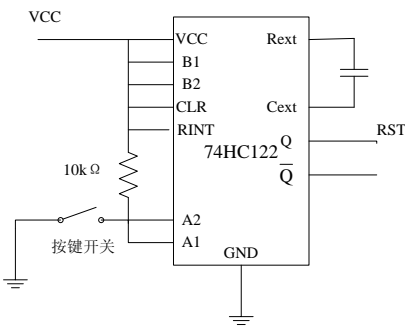


图 2-9 复杂的复位电路 2

2.6 51 系列单片机的时钟和时序

单片机内部是基于数字逻辑电路来运行的，这些数字逻辑电路需要在一个统一的时钟步调下运行，这便涉及单片机的时钟和时序问题。下面以 AT89S52 为例来介绍和时序有关的问题，其中的内容对其他 51 系列单片机都是通用的。

2.6.1 振荡器和时钟电路

振荡器和时钟电路用于产生单片机正常工作时所需要时钟信号。AT89S52 单片机采用的 CMOS 工艺，内部包含有一个振荡器，可以用于 CPU 的时钟源。另外，也允许采用外部振荡器，由外部振荡器产生的时钟信号来供内部 CPU 运行使用。下面分别介绍这两种方式。

1. 内部时钟模式

内部时钟模式是采用单片机内部振荡器来工作的模式。AT89S52 和其他 51 系列单片机一样，其内部包含一个高增益的单级反相放大器，引脚 XTAL1 和 XTAL2 分别为片内反相放大器的输入端口和输出端口，其工作频率为 0~33MHz。对于 Intel 的 8051，工作频率为 1.2~12MHz。

当单片机工作于内部时钟模式的时候，只需在 XTAL1 引脚和 XTAL2 引脚连接一个晶体振荡器或陶瓷振荡器，并接两个电容后接地即可，如图 2-10 所示。使用时，对于电容的选择有一定的要求，具体如下：

- 当外接晶体振荡器的时候，电容值一般选择 $C1=C2=30\pm 10\text{pF}$ ；
- 当外接陶瓷振荡器的时候，电容值一般选择 $C1=C2=40\pm 10\text{pF}$ 。

在实际电路设计时，应该注意尽量保证外接的振荡器和电容尽可能靠近单片机的 XTAL1 和 XTAL2 引脚，这样可以减少寄生电容的影响，使振荡器能够稳定可靠地为单片机 CPU 提供时钟信号。

2. 外部时钟模式

外部时钟模式是采用外部振荡器产生时钟信号，直接提供给单片机使用。对于不同结构的单片机，外部时钟信号接入的方式有所不同。

对于普通的 8051 单片机，外部时钟信号由 XTAL2 引脚接入后直接送到单片机内部的时钟发生器，而引脚 XTAL1 则应直接接地，如图 2-11 所示。这里需要注意，由于 XTAL2 引脚的逻辑电平不是 TTL 信号，因此建议外接一个上拉电阻。

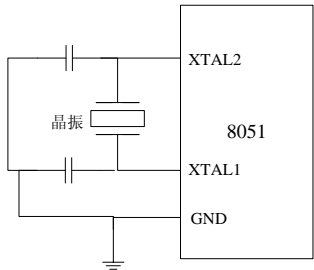


图 2-10 内部时钟模式

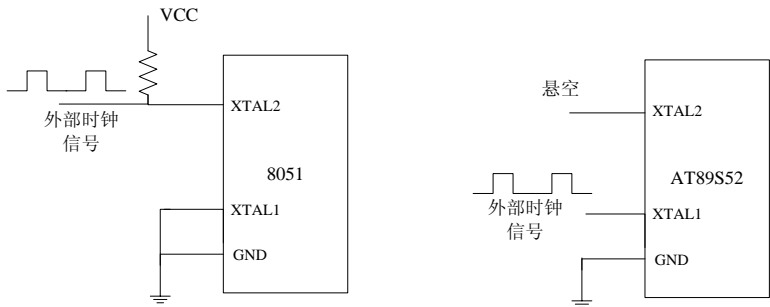


图 2-11 8051 和 AT89S52 的外部时钟模式

对于 CMOS 型的 80C51、80C52、AT89S52 等单片机，和普通的 8051 不同的是其内部的

时钟发生器的信号取自于反相放大器的输入端。因此，外部的时钟信号应该接到单片机的 XTAL1 引脚，而 XTAL2 引脚则悬空即可。

外部时钟信号的频率应该满足不同单片机的工作频率要求，比如普通的 8051 频率应该低于 12MHz，对于 AT89S52 则为 0~33MHz。如果采用其他的型号，则应具体参考该单片机的数据手册中的说明。

2.6.2 CPU 的时序

CPU 的时序是指令执行所遵从的格式。在单片机内部，振荡器始终驱动内部时钟发生器向 CPU 提供时钟信号。时钟发生器的输入是一个二分频触发器，这个二分频触发器为单片机提供了一个二相时钟信号，即相位信号 P1 和相位信号 P2，驱动 CPU 产生执行指令功能的机器周期。

单片机的时序是用定时单位来描述的，其描述了指令执行中各控制信号在时间上的关系，这里涉及节拍、状态、机器周期和指令周期 4 个概念，接下来分别说明它们之间的关系，示意图如图 2-12 所示。

- 拍 (P)：拍为振荡脉冲的周期，为方便描述，这里用 P 来表示。它是晶体的振荡周期，或者外部时钟脉冲的周期。拍是 51 系列单片机中的最小时序单元。
- 时钟周期 (S)：振荡脉冲信号经过二分频后，便可得到单片机的时钟信号，时钟信号的周期一般用 S 来表示。一个状态包含两个拍，分别称为 P1 和 P2。时钟周期是单片机 CPU 中最基本的时间单元，在一个时钟周期内，CPU 仅完成一个最基本的动作。
- 机器周期：51 系列单片机中规定，一个机器周期由 6 个时钟周期 (S1~S6) 组成，再细分可以表示为 12 个拍组成。从图 2-12 中可以看出依次为 S1P1、S1P2、S2P1、……、S6P2。如果振荡频率一旦确定，则机器周期也就确定了。比如选用 24MHz 的晶体振荡器，则对应的机器周期 T=500ns。
- 指令周期：执行一条指令所需要的时间即指令周期。不同的指令有不同的指令周期，表现为需要不同的机器周期，单周期指令执行需要一个机器周期，双周期指令执行需要两个机器周期。指令的周期一般都在 1~4 个机器周期范围内，具体可以参考指令表中的介绍。

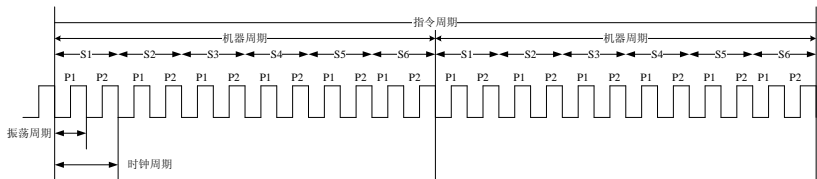


图 2-12 51 系列单片机周期的概念

2.6.3 指令执行的时序

单片机的指令执行过程包括取指令和执行指令两个部分，其是在 CPU 时钟的步调下实现的。在单片机中，不同指令的长度和指令周期一般各不相同，可以分为单字节单周期指令、双字节单周期指令、双字节双周期指令等。下面介绍几个典型的指令执行时序。

1. 单字节单周期指令的执行时序

单字节单周期指令的执行时序，如图 2-13 所示。在每个机器周期内，地址锁存信号 ALE 出现两次高电平有效信号，一次在 S1P2~S2P1，另一次在 S4P2~S5P1。这样，一个机器周期内便可以读两次程序存储器代码。

单字节单周期指令在执行时，第一次读取指令代码后便立即开始执行该指令，第二次读的代码将被丢弃，不使用。

2. 双字节单周期指令的执行时序

双字节单周期指令的执行时序，如图 2-14 所示。地址锁存信号 ALE 仍然在一个机器周期内有效两次。不同于前面，双字节单周期指令在执行时，两次读取的代码都有效，在一个机器周期内便执行完该指令。

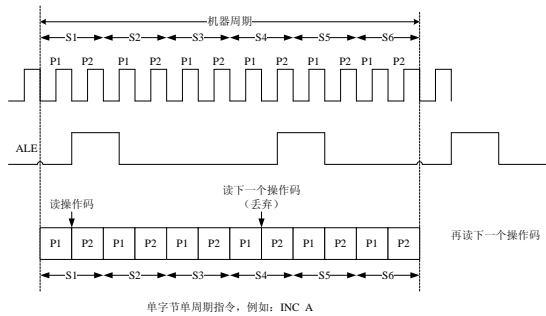


图 2-13 单字节单周期指令的执行时序

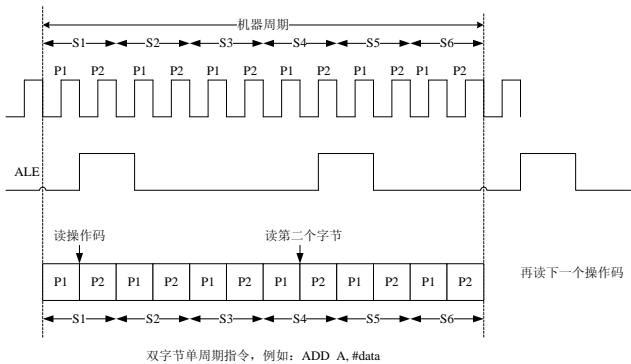


图 2-14 双字节单周期指令的执行时序

3. 单字节双周期指令的执行时序

单字节双周期指令的执行时序，如图 2-15 所示。这类指令执行时，第一次读取指令代码后，其余三次读代码操作均被丢弃，用两个机器周期执行完该指令。

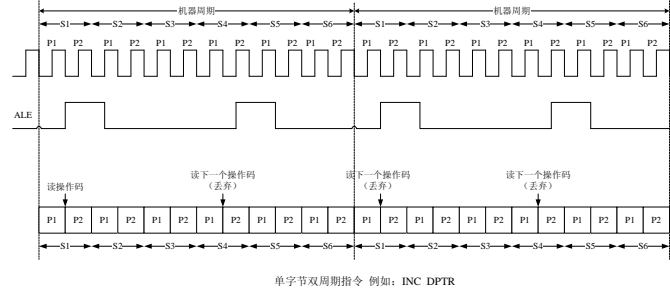


图 2-15 单字节双周期指令的执行时序

4. MOVX 类单字节单周期指令的执行时序

MOVX 类单字节单周期指令的执行时序，如图 2-16 所示。指令第一次读取代码后，第二次读代码操作被丢弃，在第二个机器周期完成外部数据单元的读写。

这里需要说明的是，无论单片机的程序是在单片机内部还是在外部存储器中，其指令读取和执行的时序都是一样的，只不过需要有其他硬件资源的配合。这将在下面一节中介绍。

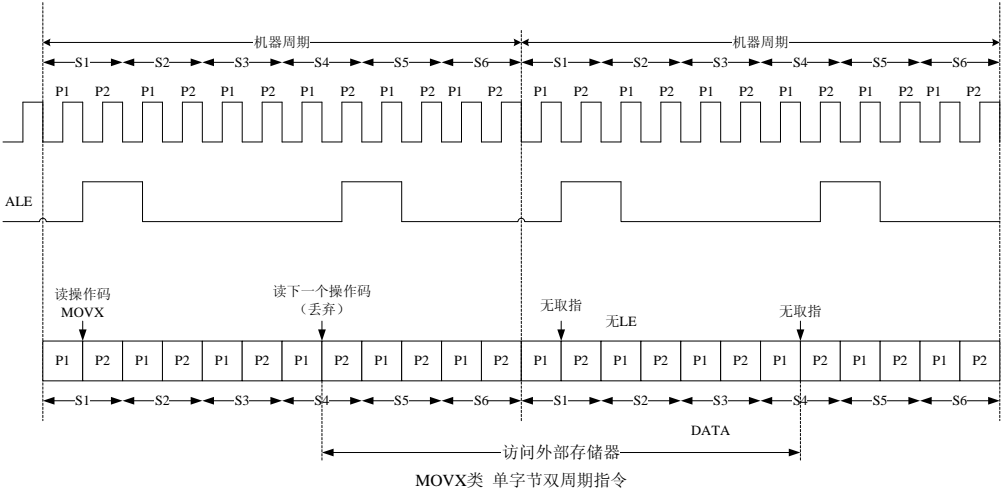


图 2-16 MOVX 类单字节单周期指令的执行时序

2.6.4 访问外部 ROM/RAM 的操作时序

访问外部 ROM/RAM 的操作时序，主要发生在外部扩展单片机 ROM 和 RAM 的时候。此时，具体的操作时序分别介绍如下。

1. 访问外部 ROM 的操作时序

当对 51 系列单片机进行外部扩展程序存储器 ROM 的时候，指令的操作时序便涉及外部存储器的操作。此时除了需要 ALE 信号外，还需要 $\overline{\text{PSEN}}$ 信号，以及将 P0 口作为低 8 位地址，P2 口作为高 8 位地址。访问外部 ROM 的时序，如图 2-17 所示。

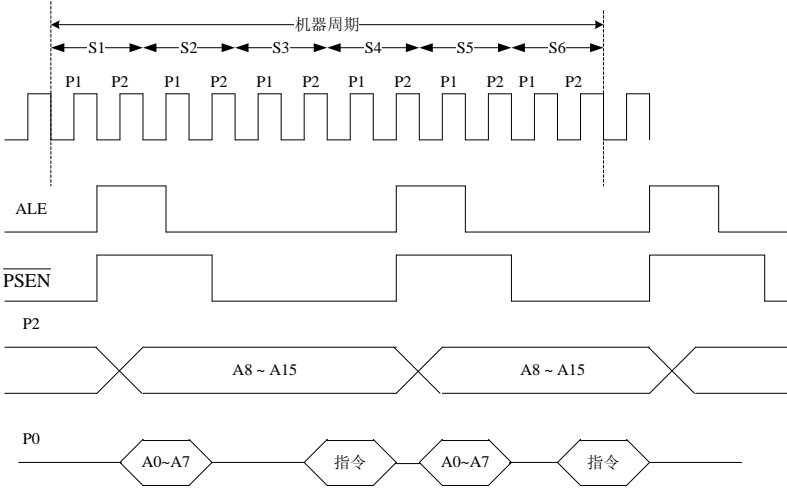


图 2-17 读外部 ROM 时序

其中 P0 端口作为地址和数据的分时复用，它先输出低 8 位地址，并利用 ALE 信号使其锁存起来，这个低 8 位地址和 P2 口的高 8 位地址共同组成 16 位地址。在 $\overline{\text{PSEN}}$ 信号有效的情况下，便可以从外部 ROM 中读取指令，再通过 P0 口送到单片机中。

2. 访问外部 RAM 的操作时序

当对单片机进行了外部 RAM 扩展, 此时单片机访问外部 RAM 的操作时序和前面有所不同。整个访问外部 RAM 的操作时序, 如图 2-18 所示。操作需要执行以下两步。

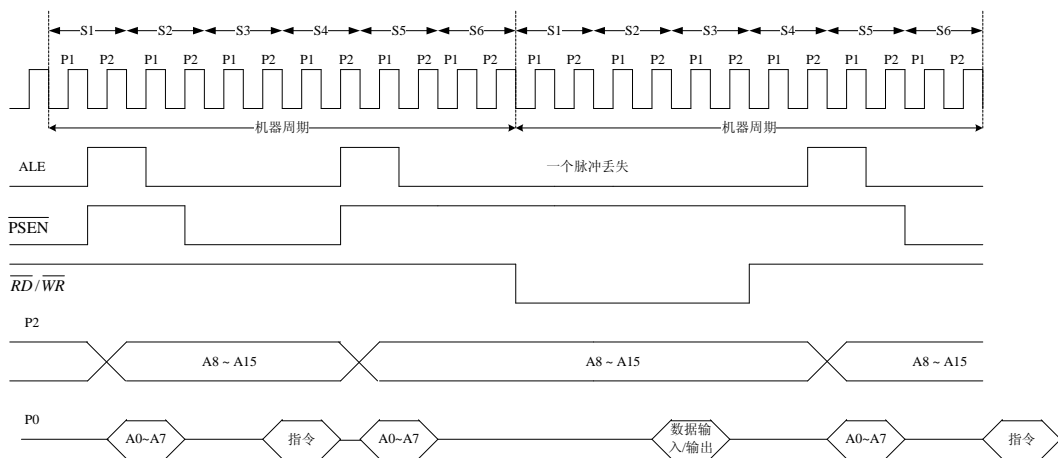


图 2-18 读写外部 RAM 时序

- (1) 先从外部 ROM 中取 MOVX 指令;
- (2) 根据 MOVX 指令所给出的数据来选择某个外部 RAM 单元, 然后对该单元进行读写操作。

第一个机器周期用来从外部 ROM 取指令。在 S4P2 之后，将取得的指令中的外部 RAM 地址送出，这时仍然是 P0 口送出低 8 位地址，P2 口送出高 8 位地址。ALE 信号用来锁存低 8 位地址。

在第二个机器周期，ALE 的第一个有效信号不再出现，而读信号 $\overline{\text{RD}}$ 有效，这样便将外部 RAM 中的数据送到单片机 P0 端口。该机器周期第二个 ALE 有效信号，没有操作进行。对于向外写操作，只需将 $\overline{\text{WR}}$ 换成 $\overline{\text{RD}}$ 即可，其余时序都不变。

2.7 51 系列单片机的并行 I/O 口

典型的 8051 单片机具有 4 个 8 位的并行 I/O 端口，分别为 P0、P1、P2 和 P3，共 32 条 I/O 线。这些 I/O 端口是双向 I/O 端口，每个端口均可以用做输入和输出。在程序中，这些 I/O 端口分别对应 4 个特殊功能寄存器 P0、P1、P2 和 P3。

2.7.1 并行 I/O 口

对于 8051 的 4 组 8 位 I/O 口来说, 其中 P1、P2 和 P3 为准双向口, P0 口则为双向三态输入输出口, 下面分别介绍这几个端口的结构。

1. P0 端口结构

P0 端口是由 8 个相同结构的引脚组成的，对于某一个引脚结构，如图 2-19 所示。P0 端口内部包含一个输出锁存器、一个输出驱动电路、一个输出控制电路、转换开关 MUX 和两个三态缓冲器，其中输出驱动电路由一对场效应管（FET）组成，整个端口的工作状态受控于输出控制电路。

P0 端口既是一个真正的双向数据总线口，也可以分时复用输出低 8 位地址总线。

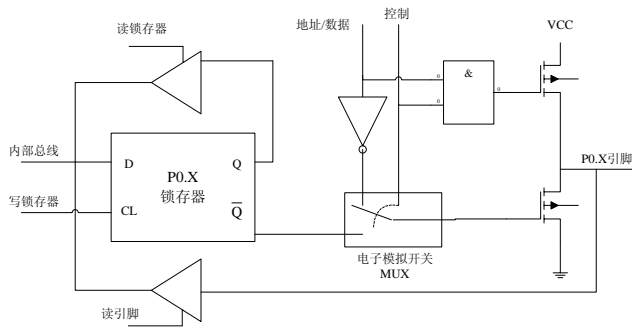


图 2-19 P0 口内部结构

当 P0 端口作为普通的 I/O 使用时，对应的控制信号为 0。电子模拟开关 MUX 将锁存器的 \bar{Q} 端和输出端连接在一起。同时与门输出为 0 使上拉 FET 管截止，这时的输出是漏极开路电路，故需要外接上拉电阻（5K Ω -10K Ω 左右）才能正常工作。其工作情况如下：

- 当程序中使该输出为 0 时，锁存器的输出端 \bar{Q} 为高电平，致使下拉 FET 管导通，从而输出端输出 0；
- 当程序中设置输出 1 时，锁存器的 \bar{Q} 为低电平，致使下拉 FET 管截止，由外接的上拉电阻将输出端变为高电平，从而输出 1。

对于输入的情况，一般应首先置各个锁存器为 1（即输出 FFH），才能保证获得正确的输入结果。即作为普通 I/O 端口时，其不是一个真正的双向 I/O 端口，而是一个准双向端口。

当 P0 口用做低 8 位地址/数据分时复用，控制信号为高电平 1，控制电子模拟开关 MUX 与地址/数据线经反相器输出相连，并于下拉 FET 导通，同时与门开锁，输出地址/数据信号，既通过与门驱动上拉 FET 管，又通过反相器驱动下拉 FET 管。其工作情况如下：

- 当输出信号 1 时，上拉 FET 管导通，而 1 经过反相器后变为 0，使下拉 FET 管截止，从而在输出引脚上输出高电平 1；
- 当输出信号 0 时，上拉 FET 管截止，而 0 经过反相器后变为 1，使下拉 FET 管导通，从而使输出引脚上输出低电平 0。

由于 P0 口做地址/数据分时复用方式时，经复位后自动置 P0 口为 0FFH，使下拉 FET 管截止，控制为 0 时上拉 FET 管也截止，从而保证在高阻抗状态下输入正确的信息。因此，P0 口作地址/数据总线时是一个真正的双向口，其能够驱动 8 个 LSTTL 负载。

2. P1 口结构

P1 口一般用做通用 I/O 端口，其可以用做位处理，各位都可以单独输出或输入信息。P1 口的结构示意图如图 2-20 所示。

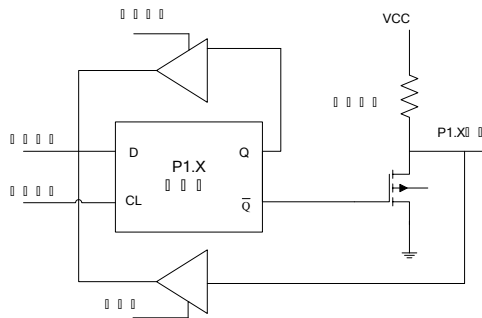


图 2-20 P1 口内部结构

P1 口同样是准双向的 I/O 端口，当需要某位先输出然后输入的时候，应该在输入操作前，加一条输出 1 的指令，然后再输入才正确。对于复位后，由于各位锁存器均置为 1， \bar{Q} 端输出为 0，下拉 FET 管截止，因此，各位用做输出或输入都是正确的。

另外，对于 AT89S52，P1 端口的某些引脚还可以有第二功能。P1.0 引脚用于定时/计数器 2 的外部事件计数输入端口，P1.1 引脚用于定时/计数器 2 的外部控制端口。P1.5~P1.7 还用于片内 Flash 的编程。

3. P2 口结构

P2 口可以当做普通 I/O 口，也可以在系统外部扩展存储器的时候，输出高 8 位的地址。P2 口单个引脚的结构示意图，如图 2-21 所示。其工作情况如下：

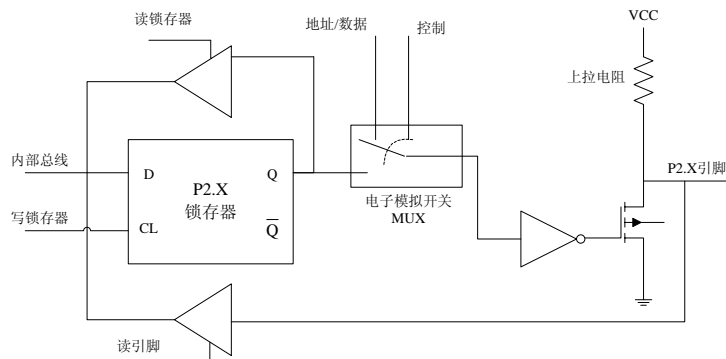


图 2-21 P2 口内部结构

- 当 P2 用做高 8 位地址时，控制信号用电子模拟开关 MUX 接通地址端，高 8 位地址信号便加到输出端口，从而实现 8 位地址的输出；
- 当 P2 用做普通 I/O 口时，控制信号用电子模拟开关 MUX 接通锁存器的 Q 端，则进行通用 I/O 操作。此时，P2 口属于准双向 I/O 口。因此，在复位情况下，直接可以从引脚输入外部数据信息；而在运行中，由输出转为输入方式的时候，则应该加一条输出 0FFH 指令，再从端口读入才正确。其余操作和 P0 口类似，P2 口可以驱动 4 个 LSTTL 负载。

4. P3 口内部结构

P3 端口是一个具有第二变异功能、且可位操作的端口。P3 口的内部结构如图 2-22 所示，其可以有以下两种用途。

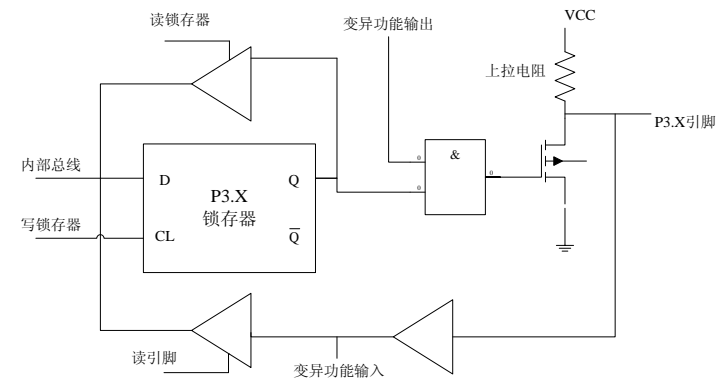


图 2-22 P3 口内部结构

51 单片机开发与应用技术详解

- 当做为普通 I/O 端口时，P3 口可以进行位操作，是准双向端口，可以驱动 4 个 LSTTL 负载。
- 当系统需要扩展外部器件时，P3 口可以作为第二变异功能使用。其各位的功能如表 2-8 所示。

表 2-8 P3 端口的第二变异功能

I/O 引脚	第二变异功能名称	功能介绍
P3.0	RXD	串行通信数据接收端口
P3.1	TXD	串行通信数据发送端口
P3.2	$\overline{\text{INT0}}$	外部中断 0 请求端口
P3.3	$\overline{\text{INT1}}$	外部中断 1 请求端口
P3.4	T0	定时/计数器 0 外部事件计数输入端
P3.5	T1	定时/计数器 1 外部事件计数输入端
P3.6	$\overline{\text{WR}}$	外部数据存储单元写选通信号
P3.7	$\overline{\text{RD}}$	外部数据存储单元读选通信号

2.7.2 并行 I/O 口的应用

单片机 4 个 8 位 I/O 端口的不同结构，决定了各自的应用范围。例如，在一些复杂的应用系统中，只用一个单片机很难达到系统的要求，经常需要外部功能扩展。因此，单片机的 P0 口和 P2 口常用于组成 16 位地址总线。P0 口用做 8 位数据总线，P3 口由于其特有的第二变异功能，因此常用于传输和控制等，只有 P1 口可以真正地用于 I/O 操作。

另外，在单片机应用时，P0 口需要外加上拉电阻，而 P1 口、P2 口和 P3 口内部设置有上拉电阻，不用外加。这 4 个 I/O 端口均为准双向 I/O 端口，其驱动能力不同，P0 口的驱动能力最强，可以驱动 8 个 LSTTL 负载，其余三个端口只能驱动 4 个 LSTTL 负载。

还有一点需要强调的是这些端口都有两种读取方式，即读锁存器和读引脚。这是因为在 I/O 端口中，锁存器内容和引脚上的内容有可能不一致。

例如，当单片机的某个 I/O 引脚用于驱动三极管的基极时，向该引脚写入 1，使三极管导通，此时基极的电压便发生由 1 到 0 的跳变，从而使该引脚也变为低电平，而此时锁存器中的内容仍然为 1。这样便出现锁存器和引脚内容不一致的情况。

在单片机的指令系统中，有些指令用来读锁存器中的内容，有些指令则用来读引脚内容。当指令的目的操作数为 I/O 口时，该指令所读的便是锁存器中的内容，而不是引脚上的内容。

读锁存器的指令，示例如下：

```
ANL P2, A
INC     P1
CPL P1.3
XRL P1, A
ORL P1, A
```

2.7.3 并行 I/O 口的扩展

虽然 51 系列单片机提供了 4 个 8 位的并行 I/O 口用于和外部设备进行数据通信及控制，但是这些 I/O 一般不能完全用于输入输出操作。

例如，很多时候都需要扩展外部程序存储器或数据存储器，此时 P0 口和 P2 口便用做数据和地址总线，提供给用户的 I/O 端口只有 P1 口和 P3 口。如果再使用串行通信或外部中断功能时，则可使用的 I/O 口便更少。因此，在单片机的系统设计中，经常需要扩展 I/O 口。

1. 并行 I/O 口的原理

51 系列单片机的 I/O 端口的扩展，一般采用将扩展的 I/O 口与外部 RAM 统一编址的方式。用户可以将外部 64KB 的 RAM 空间的一部分作为扩展 I/O 端口的地址空间，CPU 使用“MOVX”指令对此扩展 I/O 接口进行输入/输出操作，即可以像访问外部 RAM 存储单元一样来访问扩展的 I/O 端口。

并行 I/O 口的扩展有两种方式，可以采用普通的锁存器、三态门等芯片来进行简单的 I/O 口扩展，也可以采用可编程的 I/O 芯片，如 8255、8155 等来进行扩展。

可编程的 I/O 芯片比较贵，而且使用比较复杂。在实际的单片机应用系统中，一般采用第一种方法，这种 I/O 端口通过 P0 口扩展，其成本低、电路简单、使用也很方便。

2. 并行 I/O 扩展实例

这里给出使用普通的锁存器、三态门等芯片进行并行 I/O 口扩展的一个例子。电路的示意图如图 2-23 所示。

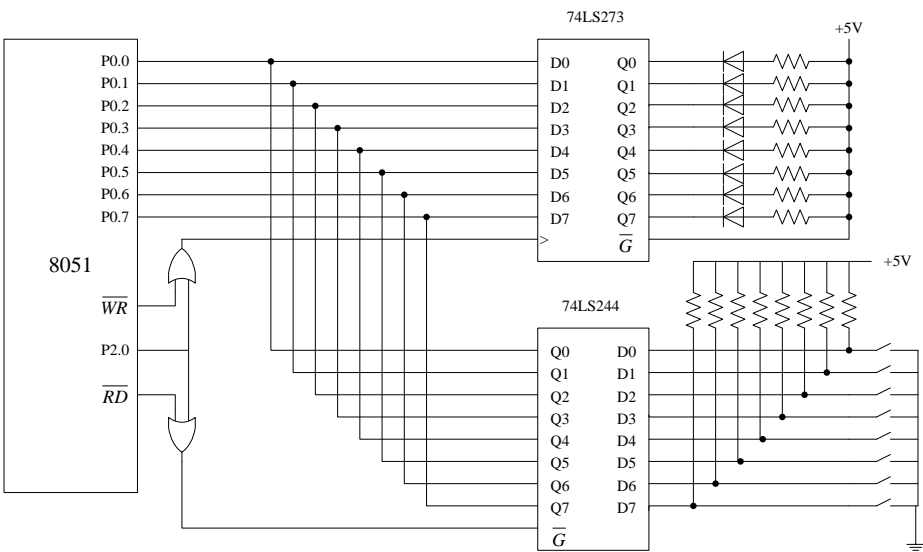


图 2-23 并行 I/O 扩展

在该电路中，采用 74LS273 作为端口的扩展输出、74LS244 作为端口的扩展输入。如果系统中还有其他扩展 I/O 端口，可以使用线选法或译码法将其所使用的地址空间分开。下面分析一下电路的原理。

在图 2-23 中，输出控制信号由引脚 P2.0 和 \overline{WR} 取或运算而得，当两者都为低电平的时候，或门输出为 0，将 P0 端口的数据锁存到 74LS273 中。74LS273 的输出控制着发光二极管，当某根引脚输出低电平的时候，对应的发光二极管发亮。

输入信号是由引脚 P2.0 和 \overline{RD} 取或运算而得，当两者都为低电平的时候，或门输出为 0，此时选通 74LS244，将外部的数据信息输入到总线。图 2-23 中，与 74LS244 相连的按键开关按下的时候，将输入 1，否则将输入 0。

可见，通过 8051 的双向数据线 P0 端口，既可以从 74LS244 输入数据，又可以将数据传送到 74LS273 中输出。

输入和输出都是在 P2.0 引脚为 0 的时候有效的，分别用 \overline{RD} 和 \overline{WR} 来区别控制的，因此不会产生输入输出冲突。

在程序中访问扩展 I/O 端口是通过“MOVX”指令来实现的。例如，如果需要读入扩展 I/O

端口的数据，并保存到累加器 A 中，则程序的示例代码如下：

```
MOV      DPTR , #0FEFFH      ;P2.0=0，即数据指针指向扩展 I/O 地址
MOVX A, @DPTR                ;从 74LS244 读入数据
```

如果需要将累加器 A 中的数据输出，则程序的示例代码如下：

```
MOV      DPTR , #0FEFFH      ;P2.0=0，即数据指针指向扩展 I/O 地址
MOVX @DPTR, A                ;向 74LS133 读出数据
```

2.8 系统掉电保护和低功耗设计

对于一个完善的单片机应用系统，为了保证其能够安全地运行及减少功耗等，需要考虑进行掉电保护和采用低功耗的工作方式。

2.8.1 掉电保护

掉电保护电路需要完成的工作是，由电压检测电路检测到电源电压下降时，触发外部中断 INT0或 INT1，在中断服务子程序中将外部 RAM 中的有用数据送入内部 RAM 保存，然后切换备用电源。

在单片机工作的时候，如果突然电源中断或电源电压不稳，都有可能造成单片机停止工作。此时，单片机 RAM 中的数据及各种寄存器的状态都将丢失。等到电源恢复的时候，单片机便重新进入新的一次执行。对于一些比较重要的系统，一般不允许。这就需要进行掉电保护，将一些重要的数据保护起来或切换到备用电源。

备用电源只为单片机内部 RAM 和专用寄存器提供维持电流，使这些重要的数据不丢失；而整个外部电路则因为电源的中断而停止工作，时钟电路也停止振荡，CPU 也停止工作。

当电源恢复的时候，备用电源还需要继续供电 10ms 左右，以保证外部电路达到稳定状态。在结束掉电保护状态时，首要的工作是将被保护的数据从内部 RAM 中恢复出来。

如图 2-24 所示是一个实用的掉电保护电路，当系统检测到电源中断的时候，立即通过外部中断输入线 INT0来中断单片机现行操作。外部中断 0 服务程序将一些重要的数据送入片内 RAM 保存，然后使 P1.0 输出低电平来触发单稳态电路 MC7555。它输出电平脉冲的宽度取决于电容和电阻的数值及电源 VCC 是否已经掉电。如果当单稳态定时输出后，若 VCC 仍然存在，这是一个假掉电报警，并从复位开始重新操作；若 VCC 已经掉电，则断电期间由单稳态电路给 RST 供电，维持片内 RAM 保存信息，一直到 VCC 恢复为止。

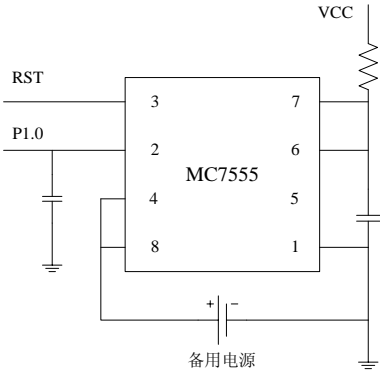


图 2-24 掉电保护电路

对于 80C51 及 AT89S52 等，掉电保护的过程不相同。当电压检测电路检测到电源电压降低的时候，同样触发外部中断，在中断服务子程序中，除了将外部 RAM 中的有用数据保存以外，还要将特殊功能寄存器的有用数据保护起来，然后对电源控制寄存器 PCON 进行设置，PCON 寄存器的各位定义如表 2-9 所示。

表 2-9 PCON 寄存器的各位定义

D7	D6	D5	D4	D3	D2	D1	D0
SMOD	-	-	-	GF1	GF0	PD	IDL

其中 SMOD 是波特率倍增位，在串口通信中使用；GF1 和 GF0 是通用标志位，由软件置位和复位；PD 是掉电控制方式，PD=1，则进入掉电方式；IDL 是待机方式控制位，IDL=1，则进入待机方式。

由软件将 PD 位设置为 1，就可以使单片机进入掉电保护状态。这是单片机的一切工作都停止，只有内部 RAM 和特殊功能寄存器的内容被保存。掉电保护的电源是通过 VCC 引脚接入的。当电源恢复正常后，系统要维持 10ms 的复位时间后才能退出掉电保护状态，复位操作将重新定义特殊功能寄存器，但内部 RAM 的内容不变，可将被保护的内容恢复。

2.8.2 低功耗设计

在野外，特别是电池供电的系统中，能源的消耗是个需要重点考虑的问题，这时希望单片机应用系统能低功耗运行，使系统的使用时间得到提高。AT89S52 单片机提供了两种低功耗工作模式：省电保持模式和休眠运行模式。下面分别进行介绍。

1. 省电保持模式

省电保持模式是将内部振荡器停止工作，此时 CPU 由于没有了时钟信号，因此内部所有的功能部件均停止工作。但是内部 RAM 和全部的特殊功能寄存器的数据将被保存。

省电保持模式和掉电保护有点类似，对于 80C51、AT89S52 等适用。当 CPU 置 PCON.1 位为 1 后，即 PD=1，系统便进入省电模式。在省电保持模式下，采用不同的程序存储器，各个端口的状态是不一样的，如表 2-10 所示列出了具体的状态。

表 2-10 省电保持模式下各端口的状态

程序存储器	ALE	$\overline{\text{PSEN}}$	P0 口	P1 口	P2 口	P3 口
内部程序存储器	0	0	数据	数据	数据	数据
外部程序存储器	0	0	浮空	数据	数据	数据

在省电保持模式下，电源电压 VCC 可以下降到 2V，但在进入省电保持模式之前 VCC 不能下降。如果需要退出省电保持模式，返回正常工作状态，则 VCC 必须首先恢复到正常的工作电压，并维持 10ms 左右的时间，确保内部振荡器稳定起振后，才可以退出省电保持模式。

退出省电保持模式可以通过硬件复位或启动外部中断两种方式执行。

- 硬件复位将使所有寄存器的内容重新初始化，但不会改变内部 RAM 中的数据。
- 通过外部中断，这个外部中断必须使系统恢复到全部进入省电保持模式之前的稳定水平。中断启动后，中断服务程序开始恢复正常操作。

2. 休眠运行模式

休眠运行模式是保持内部振荡器仍然运行，只是送往 CPU 内部的时钟信号被封锁，CPU 被冻结，但内部时钟信号仍然供给中断、串行口、定时/计数器等。CPU 内部状态，如堆栈指针 SP、程序计数器 PC、程序状态字 PSW、累加器 A 及所有其他寄存器均保持冻结前的内容

不变，各端口的状态也保持不变，片内 RAM 中的数据不丢失，外部设备也可以正常运行。

在软件中将 PCON.0 位置 1，即 IDL=1，系统便进入休眠运行模式。休眠运行模式下，主要的外部引脚状态，如表 2-11 所示。

表 2-11 休眠运行模式下各端口的状态

程序存储器	ALE	$\overline{\text{PSEN}}$	P0 口	P1 口	P2 口	P3 口
内部程序存储器	1	1	数据	数据	数据	数据
外部程序存储器	1	1	浮空	数据	地址	数据

在休眠运行模式下，电压 VCC 不能降低，但电流消耗会大大减少。系统进入休眠运行模式下，可以有两种方式退出休眠状态：硬件复位或外部中断。

- 对于硬件复位，由于在休眠运行模式下，振荡器仍然提供时钟信号，硬件复位只需要两个机器周期便可以完成。加在 RST 引脚上的复位信号可以直接将 IDL 清零，便可以退出休眠运行模式。此时，系统便从进入休眠模式的下一条指令开始继续执行下面的程序。
- 任何的中断请求，都可以将 IDL 清零，从而退出休眠运行模式。系统仍然从进入休眠模式的下一条指令开始继续执行下面的程序。通用标志位 GF0 和 GF1 可以用来指示中断是在正常模式下还是在休眠模式下发生的。

对于通用标志位的使用，需要在设置休眠运行模式的指令前，先设置通用标志位 GF0 或 GF1。当系统处于休眠运行模式时，如果被中断终止，可在中断服务例程中检测被设置的通用标志位，以便于确认系统中断是在何种情况下发生的。如果 GF0 或 GF1 为 1，则表示是在休眠状态下发生的中断。

2.9 51 系列单片机的最小系统

单片机的最小系统是单片机能够工作的最小硬件组合。对于 8051 系列单片机及其兼容的型号，其电路的最小系统大致相同，主要包括电源、晶体振荡器、复位电路等。这里以 AT89S52 为例，介绍典型的 51 系列单片机最小硬件电路，如图 2-25 所示。

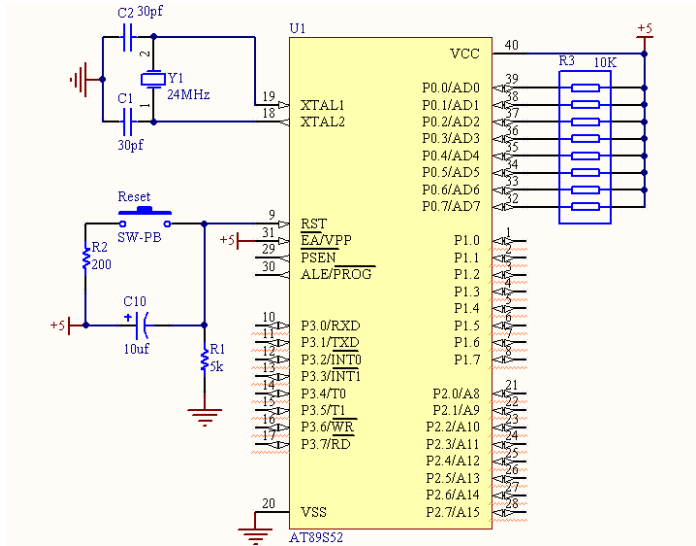


图 2-25 单片机最小系统图

这里 P0 端口接了 10K 的上拉电阻，外接晶振通过两个 30pf 的电容接地，同时采用了手动加上电复位电路。该电路可以实现复位及程序运行的基本功能，对于其他一些兼容型号的单片机同样适用。在后面的程序设计章节中，所有的程序均可以在该最小电路系统中运行。

2.10 小结

本章详细介绍了 51 系列单片机的基本结构，包括单片机的内部结构、引脚功能、中央处理器 CPU、存储器的结构、单片机的复位、单片机工作的时钟和时序、并行 I/O 端口的结构和性能，以及单片机系统的掉电保护和低功耗设计。最后，还给出了一个典型的单片机最小系统。本章在讲述的时候，不仅对基本的 8051 进行了介绍，还介绍了目前新推出的高性能单片机 AT89S52。这样读者在学习基础知识的同时，还可以了解最新单片机的性能和使用。

第 3 章 Keil C51 开发工具简介

单片机的程序设计需要在特定的编译器中进行。编译器完成对程序的编译、连接等工作，并最终生成可执行文件。对于单片机程序的开发，一般采用 Keil 公司的 μ Vision 系列的集成开发环境。 μ Vision 系列开发工具目前的最高版本是 μ Vision3，它支持汇编语言及 C51 语言的程序设计。

本章主要介绍了 μ Vision3 集成开发环境，以及如何使用 μ Vision3 集成开发环境进行单片机设计与仿真。

3.1 Keil μ Vision3 简介

Keil μ Vision 系列是德国 Keil Software 公司推出的 51 系列兼容单片机软件开发系统。 μ Vision3 是集成的可视化 Windows 操作界面，其提供了丰富的库函数和各种编译工具，能够对 51 系列单片机及和 51 系列兼容的绝大部分类型的单片机进行设计。Keil μ Vision 系列支持单片机 C51 程序设计语言，也可以直接进行汇编语言的设计与编译。

目前，Keil 公司已经被 ARM 公司收购，成为 ARM 公司旗下的产品。Keil μ Vision 系列的集成开发环境最高版本是 μ Vision3，版本号为 V8.08。Keil μ Vision 系列是非常优秀的编译器，受到广大单片机设计者的广泛使用。其主要特点如下：

- 支持汇编语言、C51 语言等多种单片机设计语言；
- 可视化的文件管理，界面友好；
- 支持丰富的产品线，除了 51 及其兼容内核的单片机外，还新增加了对 ARM 内核产品的支持；
- 具有完善的编译连接工具；
- 具备丰富的仿真调试功能，可以仿真串口、并口、A/D、D/A、定时器/计数器及中断等资源，同时也可以和外部仿真器联合进行在线调试；
- 内嵌 RTX-51 实时多任务操作系统；
- 支持在一个工作空间中进行多项目的程序设计；
- 支持多级代码优化。

3.2 μ Vision3 安装

μ Vision3 集成开发环境可以单独安装，也可以在 ARM 公司为中国区特殊设计的 ARM RealView MDK 中获得。这里以最新版的 Keil μ Vision3 V8.08 为例，来介绍其集成开发环境的安装及使用。

1. 系统要求

为了达到比较好的软件运行效果， μ Vision3 对计算机的硬件和软件配置有一定的要求，

其实这些要求是十分低的，一般的系统配置都完全可以胜任。

- 内存大于 16MB；
- 至少 45M 的硬盘剩余空间；
- Windows 95 及其以上的操作系统。

2. 软件安装步骤

μ Vision3 软件的安装操作步骤如下：

- (1) 双击 μ Vision3 的安装文件，弹出 μ Vision3 的安装界面，如图 3-1 所示。
- (2) 单击“Next”按钮，弹出“License Agreement”对话框，如图 3-2 所示。

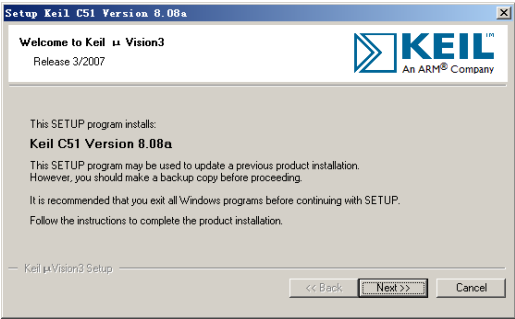


图 3-1 安装 μ Vision3

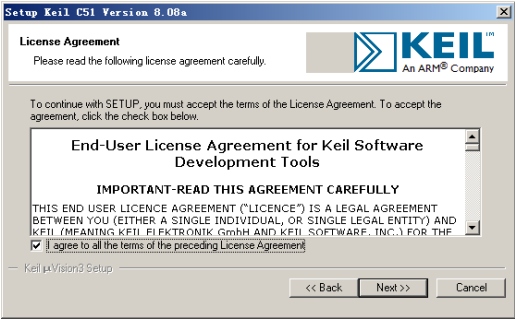


图 3-2 “License Agreement”对话框

(3) 选择接受协议，然后单击“Next”按钮，进入“Folder Selection”对话框，如图 3-3 所示。

(4) 从中选择前面的安装目录，单击“Next”按钮，此时进入用户信息输入界面，如图 3-4 所示。

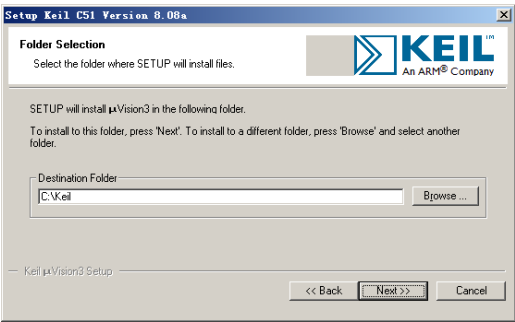


图 3-3 “Folder Selection”对话框

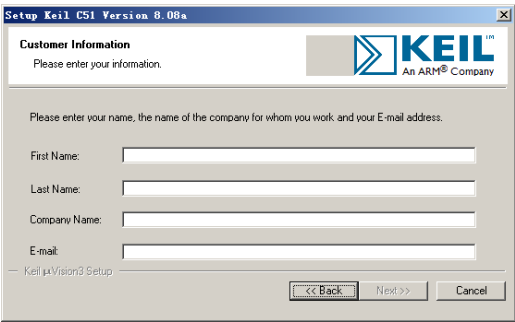


图 3-4 用户信息输入

(5) 填写完用户信息后，单击“Next”按钮，开始安装。最后弹出“Keil μ Vision3 Setup Completed”对话框，如图 3-5 所示。单击“Finish”按钮，完成 μ Vision3 的安装。

3.3 μVision3 集成开发环境

μ Vision3 集成开发环境提供了良好的图形交互界面和强大的功能，其支持绝大部分的 51 系列单片机及 ARM 内核的单片机。下面首先介绍一下 μ Vision3 的软件开发环境。

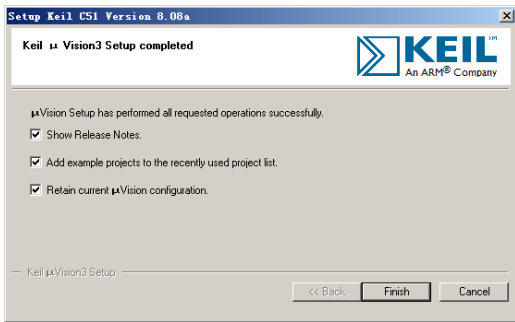


图 3-5 完成安装

3.3.1 μVision3 项目管理窗口

μ Vision3 集成开发环境是具有标准的 Windows 界面的应用程序，对于一个打开的项目工程，其界面效果如图 3-6 所示。

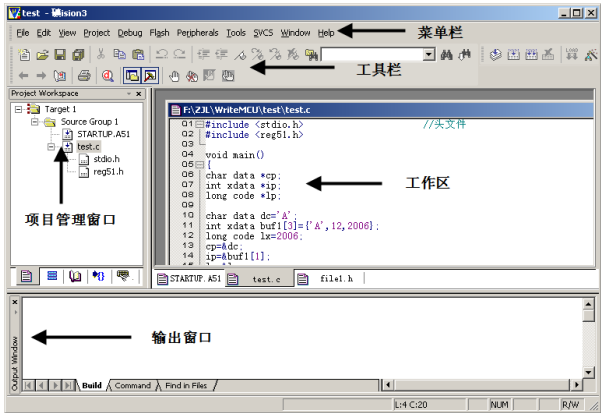


图 3-6 μ Vision3 的界面

μ Vision3 集成开发环境的界面同一般的 Windows 应用程序一样，由菜单栏、工具栏、工作区、项目管理窗口和输出窗口组成。在不同的编译环境下，可能还有其他一些调试或观察窗口，这在后面会详细介绍。

3.3.2 μ Vision3 的菜单栏

μ Vision3 的菜单栏如图 3-7 所示，其中提供了项目操作、编辑操作、编译调试及帮助等各种常用操作。下面分别介绍。

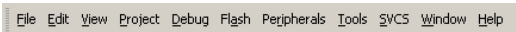


图 3-7 μ Vision3 的菜单栏

1. “File” 菜单

“File” 菜单提供了各种文件操作功能，如表 3-1 所示列出了各个菜单命令的具体功能。

表 3-1 “File” 菜单

命 令	功 能	命 令	功 能
“New” 命令	创建一个新文件	“License Management” 命令	产品注册管理
“Open” 命令	打开一个已存在的文件	“Print Setup” 命令	设置打印机
“Close” 命令	关闭当前打开的文件	“Print” 命令	打印当前文件
“Save” 命令	保存当前打开的文件	“Print Preview” 命令	打印预览
“Save as” 命令	文件另存为	“file1.c” 命令	最近打开的文件
“Save all” 命令	保存所有文件	“Exit” 命令	退出 μ Vision3
“Device Database” 命令	器件库		

“File” 菜单中的绝大多数命令和其他 Windows 应用程序中基本一致，这里就不逐个介绍了。但其中包含一个特殊的“File” → “Device Database” 命令，用于打开 CPU 器件管理对话框，如图 3-8 所示，用户可以在其中修改或添加新单片机型号的器件描述。

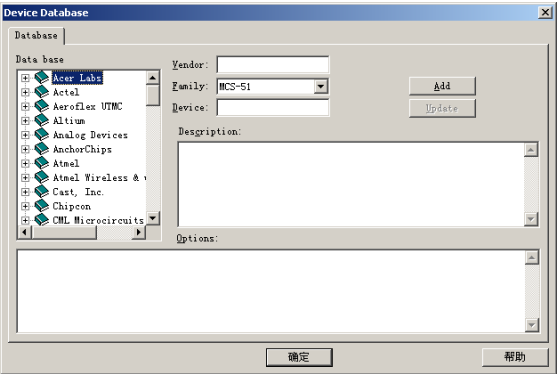


图 3-8 CPU 器件管理对话框

2. “Edit” 菜单

“Edit” 菜单提供了单片机程序源代码的各种编辑方式，如表 3-2 所示列出了各个菜单命令的具体功能。其具体的使用方法，将在后面的章节中逐步介绍。

表 3-2 “Edit” 菜单

命 令	功 能	命 令	功 能
“Undo” 命令	取消上次操作	“Goto Previous Bookmark” 命令	移动光标到上一个标签
“Redo” 命令	重复上次操作	“Clear All Bookmark” 命令	清除当前文件的所有标签
“Cut” 命令	剪切选定的文本	“Find” 命令	在当前文件中查找文本
“Copy” 命令	复制选定的文本	“Replace” 命令	替换特定的字符
“Paste” 命令	粘贴	“Find in Files” 命令	在多个文件中查找
“Indent Selected Text” 命令	将所选定的文本右移一个制表符	“Incremental Find” 命令	渐进式寻找
“Unindent Selected Text” 命令	将所选定的文本左移一个制表符	“Outlining” 命令	源代码概要
“Toggle Bookmark” 命令	设置/取消当前行的标签	“Advanced” 命令	高级编辑命令
“Goto Next Bookmark” 命令	移动光标到下一个标签	“Configuration” 命令	颜色、字体等高级配置

3. “View” 菜单

“View” 菜单提供了各种窗口和工具栏的显示和隐藏，如表 3-3 所示列出了各个菜单命令的具体功能。

表 3-3 “View” 菜单

命 令	功 能
“Status Bar” 命令	用于显示或隐藏状态条
“File Toolbar” 命令	用于显示或隐藏文件工具栏
“Build Toolbar” 命令	用于显示或隐藏编译工具栏
“Debug Toolbar” 命令	用于显示或隐藏调试工具栏
“Project Window” 命令	用于显示或隐藏项目管理窗口
“Output Window” 命令	用于显示或隐藏输出口
“Source Browser” 命令	打开资源浏览器窗口

续表

命 令	功 能
“Disassembly Window” 命令	用于显示或隐藏反汇编窗口
“Watch & Call Stack Window” 命令	用于显示或隐藏观察和堆栈窗口
“Memory Window” 命令	用于显示或隐藏存储器窗口
“Code Coverage Window” 命令	用于显示或隐藏代码报告窗口
“Performance Analyzer Window” 命令	用于显示或隐藏性能分析窗口
“Logic Analyzer Window” 命令	用于显示或隐藏逻辑分析窗口
“Symbol Window” 命令	用于显示或隐藏字符变量窗口
“Serial Window” 命令	用于显示或隐藏串口的观察窗口（包括串口#0、#1、#2）
“Toolbox” 命令	用于显示或隐藏自定义工具条
“Periodic Window Update” 命令	在程序运行时刷新调试窗口
“Include Dependencies” 命令	用于显示或隐藏包含文件

4. “Project” 菜单

“Project” 菜单提供了项目的管理和编译，如表 3-4 所示列出了各个菜单命令的具体功能。

表 3-4 “Project” 菜单

命 令	功 能
“New” 命令	创建新项目及新的工作空间
“Import μ Vision1 Project” 命令	导入 μ Vision1 的项目
“Open Project” 命令	打开一个已存在的项目
“Close Project” 命令	关闭当前项目
“Manage” 命令	定义工具、包含文件、库的路径及多项目工作空间
“Select Device for Target ‘Target 1’” 命令	为当前项目选择一个 CPU
“Remove File ‘test.c’” 命令	从当前项目中移除 test.c 文件
“Options for Target ‘Target 1’” 命令	维护一个项目的对象、文件组合文件
“Clean Target” 命令	清除编译生成的中间文件
“Build Target” 命令	编译文件并生成应用
“Rebuild all target files” 命令	重新编译所有文件并生成应用
“Batch Build” 命令	批量编译文件并生成应用
“Translate ...” 命令	编译当前文件
“Stop Build” 命令	停止编译

5. “Debug” 菜单

“Debug” 菜单提供了项目调试和仿真中使用的各种命令，如表 3-5 所示列出了各个菜单命令的具体功能。

表 3-5 “Debug” 菜单

命 令	功 能
“Start/Stop Debugging” 命令	开始/停止调试模式
“Run” 命令	运行程序，直到遇到一个断点
“Step” 命令	单步执行程序，遇到子程序则进入

续表

命 令	功 能
“Step over” 命令	单步执行程序，跳过子程序
“Step out of Current function” 命令	执行到当前函数的结束
“Run to Cursor line” 命令	执行到光标所在行
“Stop Running” 命令	停止运行程序
“Breakpoints” 命令	打开断点对话框
“Insert/Remove Breakpoint” 命令	设置/取消当前行的断点
“Enable/Disable Breakpoint” 命令	使能/禁止当前行的断点
“Disable All Breakpoints” 命令	禁止所有断点
“Kill All Breakpoints” 命令	取消所有断点
“Show Next Statement” 命令	显示下一条指令
“Debug Setting” 命令	设置调试参数
“Enable/Disble Trace Recording” 命令	使能/禁止程序运行轨迹的标识
“View Trace Records” 命令	显示程序运行过的指令
“Execution Profiling” 命令	可设置成 Off、Time 和 Call
“Setup Logic Analyzer” 命令	逻辑分析
“Memory Map” 命令	打开存储器空间配置对话框
“Performance Analyzer” 命令	打开性能分析窗口
“Inline Assembly” 命令	对某一行进行重新汇编，可以修改汇编代码
“Function Editor (Open Ini File)” 命令	编辑调试函数和调试配置文件

6. “Flash” 菜单

“Flash” 菜单提供了程序下载、擦除及配置等操作，需要外部仿真器的支持，如表 3-6 所示列出了各个菜单命令的具体功能。

表 3-6 “Flash” 菜单

命 令	功 能
“Download” 命令	下载程序
“Erase” 命令	擦除程序
“Configure Flash Tools” 命令	配置工具

7. “Peripherals” 菜单

“Peripherals” 菜单提供了单片机上的各种资源，供项目仿真调试时使用，如表 3-7 所示列出了各个菜单命令的具体功能。

表 3-7 “Peripherals” 菜单

命 令	功 能	命 令	功 能
“Reset CPU” 命令	复位 CPU	“A/D Converter” 命令	打开 A/D 转换器设置对话框
“Interrupt” 命令	打开中断设置对话框	“D/A Converter” 命令	打开 D/A 转换器设置对话框
“I/O-Ports” 命令	打开并行端口设置对话框	“I2C Controller” 命令	打开 I2C 总线控制器设置对话框
“Serial” 命令	打开串口设置对话框	“CAN Controller” 命令	打开 CAN 总线控制器设置对话框
“Timer” 命令	打开定时器设置对话框	“Watching” 命令	打开看门狗设置对话框

51 单片机开发与应用技术详解

需要指出的是，该菜单提供的单片机资源根据项目选择的 CPU 类型不同而有所变化，并不是每个类型的 CPU 都有这些命令。

8. “Tool” 菜单

“Tool” 菜单提供了第三方软件的控制，如表 3-8 所示列出了各个菜单命令的具体功能。这些第三方软件，如 PC-Lint，需要用户自行安装才能使用，μ Vision3 并没有提供。

表 3-8 “Tool” 菜单

命 令	功 能
“Setup PC-Lint” 命令	配置 Gimpel Software 的 PC-Lint 程序
“Lint” 命令	用 PC-Lint 处理当前编辑的文件
“Lint All C-Source Files” 命令	用 PC-Lint 处理项目中所有的 C 源代码文件
“Configure Socket Port Number” 命令	配置仿真端口
“Customize Tools Menu” 命令	添加用户菜单项到工具菜单中

9. “SVCS” 菜单

“SVCS” 菜单提供了软件版本的控制，如表 3-9 所示列出了各个菜单命令的具体功能。

表 3-9 “SVCS” 菜单

命 令	功 能
“Configure Version Control” 命令	配置软件版本控制系统的命令

10. “Window” 菜单

“Window”菜单提供了对窗口的排列管理，如表 3-10 所示列出了各个菜单命令的具体功能。

表 3-10 “Window” 菜单

命 令	功 能	命 令	功 能
“Cascade” 命令	以互相重叠的形式排列文件窗口	“Arrange Icons” 命令	排列主框架底部的图标
“Tile Horizontally” 命令	以不互相重叠的形式水平排列文件窗口	“Split” 命令	把当前的文件窗口分割为几个
“Tile Vertically” 命令	以不互相重叠的形式垂直排列文件窗口	“Close All” 命令	关闭所有文件窗口

11. “Help” 菜单

“Help” 菜单提供了各种帮助命令，如表 3-11 所示列出了各个菜单命令的具体功能。

表 3-11 “Help” 菜单

命 令	功 能	命 令	功 能
“μ Vision Help” 命令	μ Vision 在线帮助	“Contact Support” 命令	联系支持
“Open Books Window” 命令	打开 Keil 提供的相关书籍	“Check for Update” 命令	检查版本更新
“Simulated Peripherals for ‘AT89S52’” 命令	显示 CPU 的仿真帮助	“Tip of the Day ” 命令	打开每日提示对话框
“Internet Support Knowledgebase” 命令	因特网上的帮助信息	“About μ Vision” 命令	显示软件的版本信息和许可证信息

以上列出了 μ Vision3 中所有的菜单命令，通过这些菜单命令，用户可以完成μ Vision3 相关的所有操作。用户需要在实际学习中逐步使用，以便熟悉各个命令。

3.3.3 μ Vision3 的工具栏

同其他的 Windows 应用程序一样，μ Vision3 除了在菜单栏提供了完整而丰富的操作命令外，也提供了相当完善的工具栏，便于用户快速进行操作。下面分别进行介绍。

1. 文件操作工具栏

文件操作工具栏提供了各种源文件的操作，如图 3-9 所示。如表 3-12 所示列出了每个工具栏图标对应的菜单命令，以及对应的功能描述。



图 3-9 文件操作工具栏

表 3-12 文件操作工具栏图标及功能

工具栏图标	对应的菜单命令	功能描述
	“File” → “New”	创建一个新文件
	“File” → “Open”	打开一个已存在的文件
	“File” → “Save”	保存当前打开的文件
	“File” → “Save All”	保存所有文件
	“Edit” → “Cut”	剪切选定的文本
	“Edit” → “Copy”	复制选定的文本
	“Edit” → “Paste”	粘贴
	“Edit” → “Undo”	取消上一次操作
	“Edit” → “Redo”	重复上一次操作
	“Edit” → “Indent Selected Text”	将所选定的文本右移一个制表符
	“Edit” → “Unindent Selected Text”	将所选定的文本左移一个制表符
	“Edit” → “Toggle Bookmark”	设置/取消当前行的标签
	“Edit” → “Goto Next Bookmark”	移动光标到下一个标签
	“Edit” → “Goto Previous Bookmark”	移动光标到上一个标签
	“Edit” → “Clear All Bookmarks”	清除当前文件的所有标签
	“Edit” → “Find in Files”	在多个文件中查找
		查找文本
	“Edit” → “Find”	在当前文件中查找文本
	“Edit” → “Incremental Find”	渐进式寻找
		转向前一个位置
		转向新的位置
	“View” → “Source Browser”	打开资源浏览器窗口
	“File” → “Print”	打印当前文件
	“Debug” → “Start/Stop Debugging”	开始/停止调试模式
	“View” → “Project Window”	用于显示或隐藏项目管理窗口
	“View” → “Output Window”	用于显示或隐藏输出窗口
	“Debug” → “Insert/Remove Breakpoint”	设置/取消当前行的断点
	“Debug” → “Kill All Breakpoints”	取消所有断点
	“Debug” → “Enable/Disable Breakpoint”	使能/禁止当前行的断点
	“Debug” → “Disable All Breakpoints”	禁止所有断点

2. 编译工具栏

编译工具栏提供了编译项目和文件的各种操作，如图 3-10 所示。其中的每个工具栏图标和对应的菜单命令，以及功能描述如表 3-13 所示。



图 3-10 编译工具栏

表 3-13 编译工具栏图标及功能

工具栏图标	对应的菜单命令	功能描述
	“Project” → “Translate ...”	编译当前文件
	“Project” → “Build Target”	编译文件并生成应用
	“Project” → “Rebuild all target files”	重新编译所有文件并生成应用
	“Project” → “Stop Build”	停止编译
	“Flash” → “Download”	下载程序
	“Project” → “Options for Target ‘Target 1’”	维护一个项目的对象、文件组和文件
		选择目标项目
	“Project” → “Components, Environment, Books”	定义工具、包含文件和库的路径
	“Project” → “Manage” → “Multi-project Workspace”	管理多项目工作空间
	“Edit” → “Configuration”	颜色、字体等高级配置

3. 调试工具栏

调试工具栏提供了项目仿真和调试过程中经常使用的命令，如图 3-11 所示。各个工具栏图标和对应的菜单命令，以及功能描述如表 3-14 所示。



图 3-11 调试工具栏

表 3-14 调试工具栏图标及功能

工具栏图标	对应的菜单命令	功能描述
	“Peripherals” → “Reset CPU”	复位 CPU
	“Debug” → “Run”	运行程序，直到遇到一个断点
	“Debug” → “Stop Running”	停止运行程序
	“Debug” → “Step”	单步执行程序，遇到子程序则进入
	“Debug” → “Step over”	单步执行程序，跳过子程序
	“Debug” → “Step out of Current function”	执行到当前函数的结束
	“Debug” → “Run to Cursor line”	执行到光标所在行
	“Debug” → “Show Next Statement”	显示下一条指令
	“Debug” → “Enable/Disable Trace Recording”	使能/禁止程序运行轨迹的标识
	“Debug” → “View Trace Records”	显示程序运行过的指令
	“View” → “Disassembly Window”	显示/隐藏反汇编窗口
	“View” → “Watch & Call Stack Window”	显示或隐藏观察和堆栈窗口
	“View” → “Code Coverage Window”	用于显示或隐藏代码报告窗口
	“View” → “Serial Window #1”	显示或隐藏串口 1 的观察窗口
	“View” → “Memory Window”	显示或隐藏存储器窗口

续表

工具栏图标	对应的菜单命令	功能描述
	“View” → “Performance Analyzer Window”	显示或隐藏性能分析窗口
	“View” → “Logic Analyzer Window”	显示或隐藏逻辑分析窗口
	“View” → “Symbol Window”	显示或隐藏符号窗口
	“View” → “Toolbox”	显示或隐藏自定义工具条

3.3.4 μ Vision3 的管理配置

μ Vision3 的集成开发环境提供了良好的项目管理配置，用户可以根据自己的习惯和需要进行适当的配置。

选择“Edit”→“Configuration”命令，此时弹出“Configuration”对话框，如图 3-12 所示。其中有多选项卡，下面分别对其进行介绍。每个选项卡中有很多设置选项，这里不再逐个介绍，仅选择其中最常用的进行介绍。

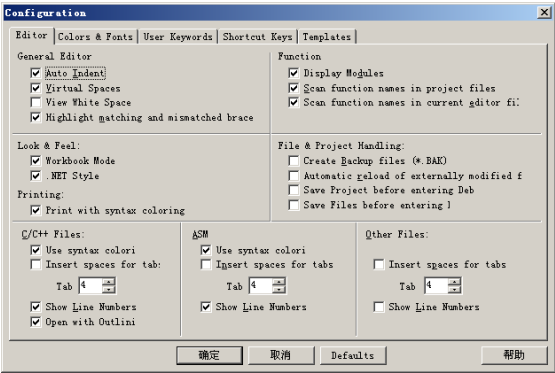


图 3-12 “Configuration”对话框

1. “Editor”选项卡

“Editor”选项卡中可以设置源代码编辑窗口的各种配置参数。其中几个主要的选项叙述如下：

- “Auto Indent”复选框：选中该复选框，则在编辑源代码文件的时候，自动以 Tab 键的距离进行缩进。
- “Create Backup files (*.BAK)”复选框：选中该复选框，则在源文件编辑过程中自动产生备份文件。
- “Use Syntax Color in ...”复选框：该复选框在“C/C++ Files”设置栏和“ASM”设置栏下各有一个。选中该复选框，则在编辑源代码文件的时候，系统自动以默认的颜色来显示 C/C++和 ASM 源代码的各种句法。



2. “Colors & Fonts”选项卡

“Colors & Fonts”选项卡中可以设置各个窗口显示的颜色方案，如图 3-13 所示。

- “8051: Editor Asm Files”用于设置 ASM 源文件中的各种关键字和语法等的颜色方案。
- “8051: Editor C Files”用于设置 C 源文件中的各种关键字和语法等的颜色方案。
- “Build Output Window”用于设置编译输出窗口的颜色方案。
- “Debug Command Window”用于设置调试命令窗口中的颜色方案。
- “Disassembly Window”用于设置反汇编窗口中的颜色方案。
- “Editor Text Files”用于设置文本编辑器中的颜色方案。

- “Logic Analyzer” 用于设置逻辑分析窗口中的颜色方案。
- “Memory Window” 用于设置存储器窗口中的颜色方案。
- “RT-Agent Window” 用于设置实时多任务窗口中的颜色方案。
- “Serial #1 Window” 用于设置串行口 1 窗口中的颜色方案。
- “Serial #2 Window” 用于设置串行口 2 窗口中的颜色方案。
- “Serial #3 Window” 用于设置串行口 3 窗口中的颜色方案。

3. “User Keywords” 选项卡

“User Keywords” 选项卡中可以设置一些自定义的关键词，如图 3-14 所示。选择关键词的作用范围，然后单击  按钮，即可新建一个关键词。单击  按钮，可以删除一个自定义关键词。

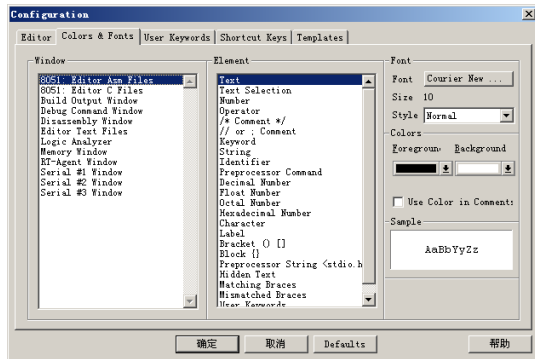


图 3-13 “Colors & Fonts” 选项卡

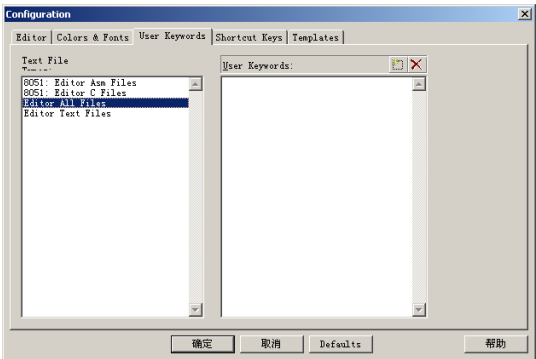


图 3-14 “User Keywords” 选项卡

4. “Shortcut Keys” 选项卡

“Shortcut Keys” 选项卡中列出了每个菜单命令的快捷键，如图 3-15 所示。用户可以根据自己的习惯，自定义某些操作的快捷键。

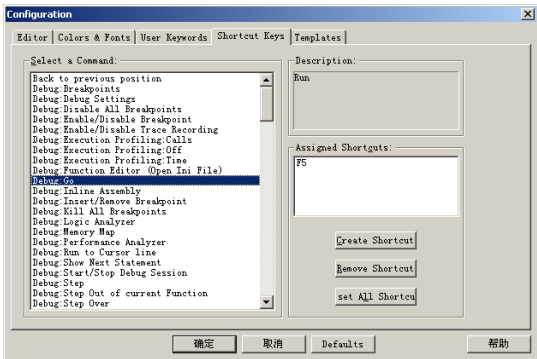


图 3-15 “Shortcut Keys” 选项卡

下面列出一些常用的系统默认快捷键，如表 3-15 所示。



表 3-15 常用的系统默认快捷键

操作类型	快 捷 键	描 述
文件操作快捷键	Ctrl+N	创建新文件
	Ctrl+O	打开已经存在的文件
	Ctrl+S	保存当前文件
	Ctrl+P	打印当前文件

续表

操作类型	快 捷 键	描 述
源代码操作快捷键	F2	移动光标到下一个标签处
	F3	向前重复查找
	Shift+F2	移动光标到上一个标签处
	Shift+F3	向后重复查找
	Ctrl+F3	查找光标处的单词
	Ctrl+F	在当前文件中查找文本
	Ctrl+Z	取消上次操作
	Ctrl+Shift+Z	重复上次操作
	Ctrl+X	剪切所选文本
	Ctrl+Y	剪切当前行的所有文本
	Ctrl+C	复制所选文本
	Ctrl+V	粘贴
	Ctrl+F2	设置/取消当前行的标签
	Ctrl+H	替换特定的字符
项目操作快捷键	Alt+F7	设置对象、组或文件的工具选项
	F7	编译修改过的文件并生产应用
	Ctrl+F7	编译当前文件
调试快捷键	F5	运行程序，直到遇到一个中断
	Ctrl+F5	开始/停止调试模式
	F11	单步执行，遇到子程序则进入
	Ctrl+F11	执行到当前函数的结束
	F10	单步执行，跳过子程序
	Esc	停止程序运行

5. “Templates” 选项卡

“Templates” 选项卡中列出了一些语句的模板结构，如图 3-16 所示。用户在不清楚某些语法的情况下，可以直接调用这些模板结构，也可以单击  按钮，新建模板结构；还可以单击  按钮，删除一个模板结构。

熟练使用这些管理工具，可以使 μ Vision3 集成开发环境更加适合个人的操作习惯，从而提高程序开发的效率。

3.3.5 μ Vision3 的各种常用窗口

μ Vision3 集成开发环境中提供了很多不同用途的窗口，利用这些窗口可以完成源代码的编辑、反汇编的查看、各种编译和调试的输出结果、堆栈中的数据查看、程序变量的内容查看，以及仿真波形图等。下面介绍一些在程序设计及仿真调试中常用的窗口及操作。

1. 源代码编辑窗口

源代码编辑窗口用于编辑程序的源代码，

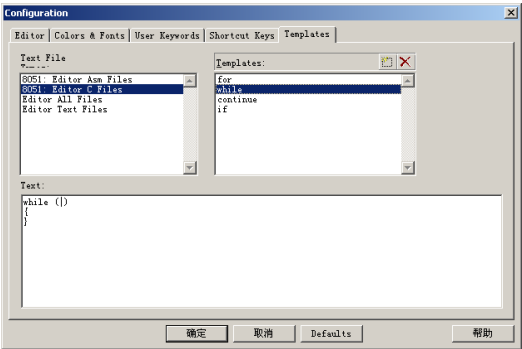


图 3-16 “Templates” 选项卡

如图 3-17 所示。

下面介绍源代码编辑窗口中几个常用的操作。

➤ 设置标签

在源代码编辑窗口中，标签可以设置在任意行，使用标签可以快速地查找和定位文本。这一功能在源代码比较长的时候十分有用。

将光标放置在需要设置标签的行，选择“Edit”→“Toggle Bookmark”命令，或者单击工具栏图标可以设置或取消当前行的标签。

➤ 标签之间的转移

标签之间的切换十分方便，单击工具栏图标，可以将光标移到下一个标签处；单击，则可以将光标移动到上一个标签处。

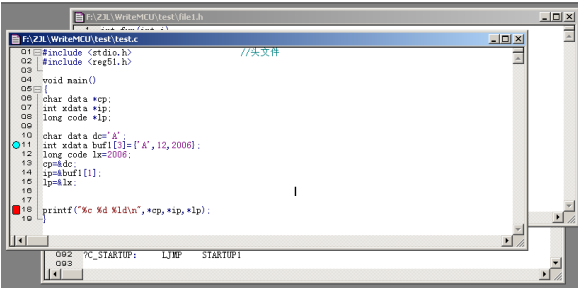


图 3-17 源代码编辑窗口

2. 反汇编窗口

反汇编窗口在程序运行或调试状态下才会出现，如图 3-18 所示。选择“Debug”→“Start/Stop Debug Session”命令，进入调试模式，此时可以通过选择“View”→“Disassembly Window”命令来显示或隐藏反汇编窗口。

反汇编窗口中列出了当前 C51 程序的反汇编结果。选择“Debug”→“Enable/Disable Trace Recording”命令后，可以跟踪指令执行的历史记录，已经执行的指令都可以用“Debug”→“View Trace Records”命令来显示。

3. 观察和堆栈窗口

观察和堆栈窗口也是在程序运行和调试状态才有的，如图 3-19 所示。选择“Debug”→“Start/Stop Debug Session”命令，进入调试模式，此时可以通过选择“View”→“Watch & Call Stack Window”命令来显示或隐藏观察和堆栈窗口。

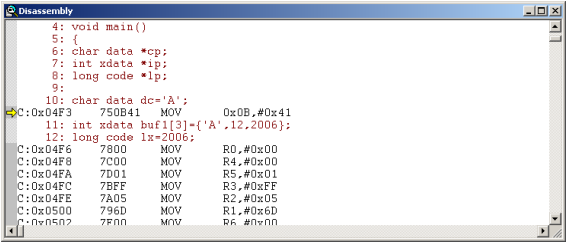


图 3-18 反汇编窗口

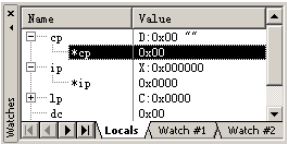


图 3-19 观察和堆栈窗口

观察和堆栈窗口中有三个选项页，“Locals”页显示在程序执行过程中，正在执行的函数里面所有的局部变量。在“Watch”页中，可以自行编辑需要观察的变量，以便于程序的调试。编辑的方法可以采用如下几种方式。

➤ 在“Watch #1”或“Watch #2”窗口中，单击“<Type F2 to edit>”选中文字，然后按

F2 键便可以进入变量编辑状态，直接输入需要观察的变量名即可。

- 进入调试状态后，在源代码窗口，右键单击需要观察的变量，选择“Add to Watch Window”命令，将该变量添加到一个“Watch”窗口中。

观察和堆栈窗口中还有一个“Call Stack”选项页，用于显示当前函数的调用情况。

4. 存储器窗口

存储器窗口也是在程序运行和调试状态才有的，如图 3-20 所示。选择“Debug”→“Start/Stop Debug Session”命令，进入调试模式，此时可以通过选择“View”→“Memory Window”命令来显示或隐藏存储器窗口。

存储器窗口提供了 4 个不同的存储器显示页，可以用不同的显示页显示不同的存储器内容或不同地址的存储器内容。例如可以分别显示内部数据存储器、外部数据存储器和代码存储器中的内容，存储区域也可以由用户自行划分。

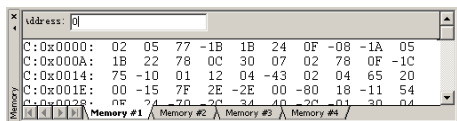


图 3-20 存储器窗口

在存储器窗口的“Address”文本框中，可以输入地址值，或者表达式来查看某个寄存器单元中的内容。

这里需要注意的是，存储器窗口的内容一般只在程序执行到断点或程序停止后才能显示。如果需要程序运行过程中显示，可以选择“View”→“Periodic Window Update”命令，这样，存储器窗口的内容便随程序的执行而周期性地显示。

5. CPU 寄存器窗口

CPU 寄存器窗口在程序运行和调试状态下显示，如图 3-21 所示。可以单击项目管理窗口下的“Regs”标签来显示。CPU 寄存器窗口显示了 CPU 寄存器中的值，其中的值随着程序的执行而不断变化。

6. 串行窗口

串行窗口只在程序运行或调试状态下显示。选择“Debug”→“Start/Stop Debug Session”命令，进入调试模式，此时可以通过选择“View”→“Serial Window #1”命令来显示或隐藏串行窗口 1。如图 3-22 所示显示的是一个示例程序运行的时候，在串行窗口中的输出内容。

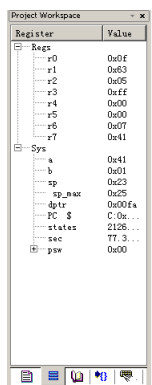


图 3-21 CPU 寄存器窗口

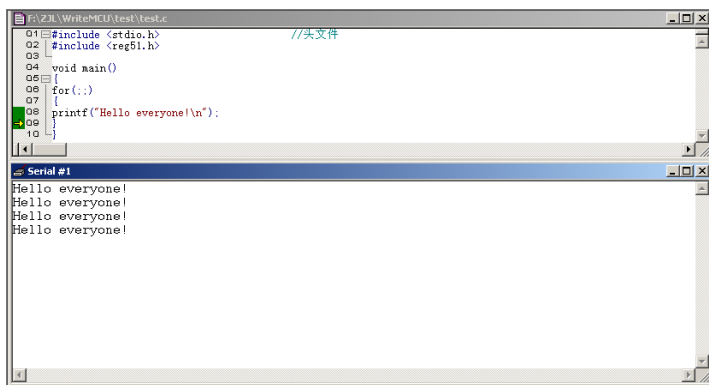


图 3-22 串行窗口

7. 逻辑分析窗口

逻辑分析窗口只在程序运行或调试的时候显示，如图 3-23 所示。其中显示的是程序运行时，各个变量的波形图。

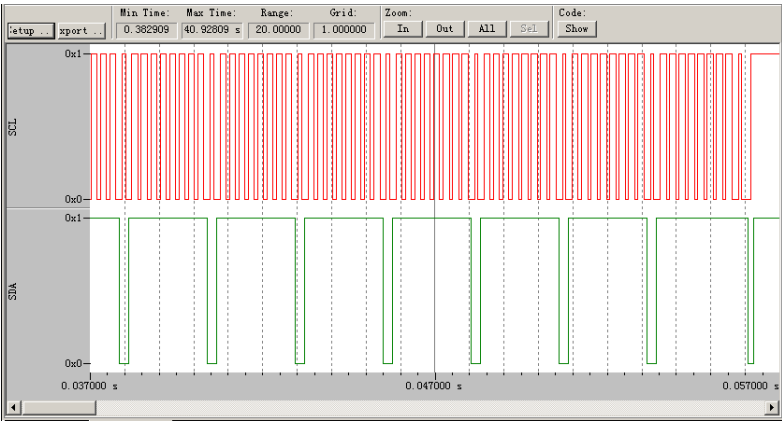




图 3-23 逻辑分析窗口

可以单击“Setup”按钮，此时弹出“Logic Analyzer”对话框，如图 3-24 所示。可以分别通过单击  按钮和  按钮在其中添加或删除需要观察的变量。

8. 符号观察窗口

符号观察窗口也需要在程序调试或运行时才能显示，如图 3-25 所示。用户可以选择“View”→“Symbol Window”命令来显示或隐藏符号观察窗口。符号观察窗口显示了程序中所有函数和模块的公共符号、当前模块或函数的局部符号、代码行号和当前载入应用所定义的 CPU 特殊功能寄存器 SFR。

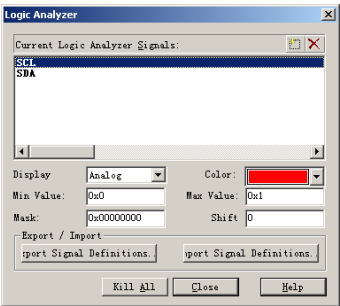


图 3-24 “Logic Analyzer”对话框

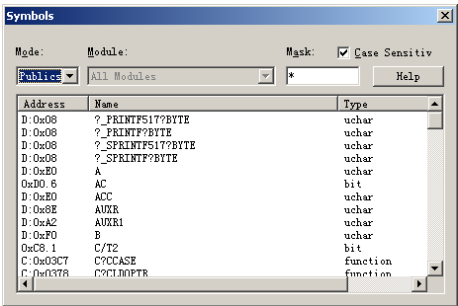


图 3-25 符号观察窗口

3.4 小结

本章详细介绍了 Keil μ Vision3 的集成开发环境 μ Vision3，包括安装过程、项目管理窗口、菜单栏、工具栏，以及 μ Vision3 的管理配置和各种常用的窗口。 μ Vision3 是一个十分优秀的单片机开发软件，应用得十分广泛，熟练掌握 μ Vision3 集成开发环境的使用是单片机设计的基础。

第二篇 编程篇——汇编语言

第 4 章 汇编语言程序设计

汇编语言是最早应用于单片机开发与应用的程序语言。相比其他程序设计语言，汇编语言执行速度快、代码短小精悍，且指令的执行周期确定。但是汇编语言也有不足之处，比如其指令复杂、缺乏通用性、不便于程序的移植等。目前，随着电子技术的发展，汇编语言的使用范围越来越小，逐渐被 C51 语言所代替，但是学习汇编语言对理解 8051 的结构及指令是很有帮助的。

本章将主要介绍汇编语言的概述、指令格式，以及汇编语言的程序结构。

4.1 汇编语言程序概述

汇编语言程序是单片机汇编指令的集合。汇编语言程序设计与 51 系列单片机的汇编指令集和硬件结构等有密切联系，其在单片机上运行可以实现特定的功能和任务。汇编语言以其简练的代码、快速的操作硬件能力而获得了广泛使用，下面首先介绍汇编语言的基本情况。

4.1.1 汇编语言简介

汇编语言是采用助记符描述指令的程序设计语言，其中助记符标识是二进制指令的形象标记。由于现在采用的计算机都是二进制的，其所能执行的每一条指令都对应一组二进制代码。采用二进制代码来表示指令和数据的语言称为机器语言，它是唯一能被计算机识别的语言；然而这种语言很难记忆和辨认，给程序设计工作带来很大麻烦。为了容易理解和记忆这些计算机的指令，人们用一些英文单词和字符作为助记符来描述每一条二进制指令的功能。用助记符标识的指令就是计算机的汇编语言，汇编语言和机器语言是一一对应的。

采用汇编语言编写的程序，每条指令的含义一目了然，这给单片机程序的编写、阅读和修改带来很大方便；而且用汇编语言编写的程序代码少，执行速度快，每条指令的执行时间完全确定。因此，在最初的单片机应用系统中，程序主要是用汇编语言来编写。

总结起来，汇编语言与高级语言相比具有以下优点。

- 占用比较少的内存单元和 CPU 资源；
- 程序代码简短，执行速度快；
- 和硬件结构密切相关，可直接调用计算机的全部资源，并可有效地利用计算机的专有特性；
- 能准确地掌握指令的执行时间，适用于实时控制系统。

当然，汇编语言也有它的缺点，比如缺乏通用性、程序不易移植、代码难懂等；而且其仍然是一种面向机器的低级语言。使用汇编语言编写程序时，必须熟悉机器或单片机的指令系统、寻址方式、寄存器的设置和使用方法，以及系统的硬件资源等。每个计算机系统都有它自己的

汇编语言指令集，不同计算机或单片机的汇编语言之间一般不能通用。但是掌握了一种计算机的汇编语言，有助于学习其他计算机或单片机的汇编语言。

4.1.2 汇编语言程序设计步骤

使用汇编语言进行计算机或单片机程序设计的流程如图 4-1 所示，一般可以分为以下几个步骤。

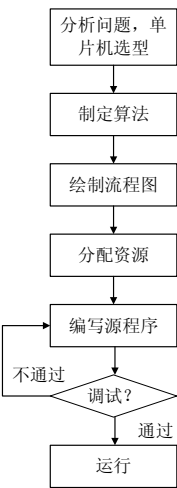


图 4-1 单片机汇编语言程序设计流程图

(1) 分析问题，单片机选型。根据实际问题的要求，初步确定程序的规模，以及需要的资源，从而选择合适的单片机型号用来进行整个系统的设计。

(2) 制定算法。根据系统需要实现的功能，确定其中使用的公式及运算流程，并转化成汇编语言的算法形式。

(3) 绘制程序的流程图。根据算法和系统的功能组成，绘制单片机系统工作的流程，以便于更好地规划整个程序的执行步骤。

(4) 分配资源。根据系统工作的需要，合理选择并分配内存单元及工作寄存器。

(5) 编写源程序。当整个系统的工作流程及单片机资源分配分析完毕后，便可以根据绘制的流程图，开始编写汇编语言的源程序代码。

(6) 上机调试程序。程序编写完毕，或者某个子程序完成后，在编译仿真软件中进行编译调试，检查并修改程序中的错误，并分析程序执行的结果是否和预期的一致，直至正确为止。

(7) 在单片机上运行。程序仿真调试通过后，便可以将程序下载到单片机芯片内，并在实际硬件电路中执行程序。

4.1.3 汇编语言程序实例

首先，为了让读者认识一下汇编语言的结构，这里举一个汇编语言的例子。程序示例如下：

```
ORG      0200H      ;汇编程序起始指令
MOV      A,  32H     ;将单字节数据存入累加器 A 中
MOV      B,  #100    ;分离出百位数
DIV      AB
MOV      R5, A       ;百位数送入寄存器 R5
XCH      A,  B       ;余数存入累加器 A 中
MOV      B,  #10     ;分离出十位和个位
DIV      AB
SWAP A              ;十位存入高字节
ADD      A,  B       ;低位存入低字节中
MOV      R6, A
END          ;汇编程序结束命令
```

这段程序主要用于将一个单字节十六进制数转换成 BCD 码的形式。单字节十六进制数据在 0~255 之间，将其除 100 后，商为百位数，余数除以 10，商为十位数，余数为个位数。这里设单字节存放在单片机 RAM 的 32H 中，转换后，百位数存放于 R5 中，十位和个位分别存放于 R6 的高位和低位字节中。

从上面示例中可以看出，汇编语言的指令采用的是单片机的指令集，指令的用法在后面的指令系统中会详细介绍。下面主要介绍如何利用这些指令进行汇编语言的程序设计及汇编程序的结构。

汇编语言的程序一般是由指令和伪指令两部分构成的。其中，51 系列单片机的各种指令及其功能将在后面的指令系统中详细介绍，这里主要介绍一下单片机系统的伪指令。

4.2 伪指令

伪指令是为汇编程序提供某种信息的指令。伪指令不能命令 CPU 执行某种操作，也没有对应的机器代码，而 51 系列单片机的指令能使单片机的 CPU 执行某种操作，能生成对应的机器代码。51 系列单片机汇编程序常用的伪指令有如下几类。

4.2.1 汇编程序起始伪指令 ORG

汇编程序起始伪指令 ORG 主要用于规定汇编程序块或数据块存放的起始地址，它的使用格式如下：

```
[标号:] ORG 16 位地址
```

示例如下：

```
ORG 0200H
START: MOV A, #45H
.....
```

这个例子中，伪指令 ORG 规定了程序的第一条指令从地址 0200H 单元开始存放，即标号 START 的值为 0200H。

在一个汇编语言源程序的开始部分，通常都要放置一条 ORG 伪指令来指定该程序代码在存储器中存放的起始地址。若省略 ORG 伪指令，则程序代码将从 0000H 单元开始存放。

！注意：在一个汇编语言源程序中，可以多次使用 ORG 伪指令，以规定不同程序段或数据段存放的起始地址，但所规定的地址应该是由小到大排序，而且绝对不允许有重叠区域。

4.2.2 汇编程序结束伪指令 END

汇编程序结束伪指令 END 是汇编语言源程序的结束标志。END 伪指令的使用格式如下：

```
[标号:] END [表达式]
```

在程序中，END 之后所写的任何指令，汇编程序都不予以处理。一个源程序只能有一个 END 伪指令。对于同时包含有主程序和子程序的汇编源程序中，同样也只能有一个 END 伪指令。

4.2.3 等值伪指令 EQU（或=）

等值伪指令 EQU 是将一个数或特定的汇编符号赋予规定的字符名称。它的使用格式如下：

```
字符名称 EQU 数据或汇编符号
```

或者如下：

```
字符名称 = 数据或汇编符号
```

用 EQU 指令赋值后的字符名称可以用做数据地址、代码地址、位地址或当做一个立即数来使用。因此，给字符名称所赋的值可以是 8 位数，也可以是 16 位数。需要注意的是，这里使用的“字符名称”不是标号，不能用“:”来做分隔符。

等值伪指令 EQU 的使用示例代码如下：

```
DX EQU 32
DA EQU 2028H
MOV A, DX
LCALL DA
```

.....

在这段程序中，DX 赋值后当做直接地址使用，而 DA 则被定义为 16 位地址，作为一个子程序的入口被调用。

使用等值伪指令 EQU 时，需要注意以下事项。

- 使用伪指令 EQU 时必须先赋值后使用，而不能先使用后赋值；
- 用 EQU 定义的字符名称不能和汇编语言的关键字同名；
- 用 EQU 定义的符号不允许重复定义，而“=”定义的符号允许重复定义。

4.2.4 数据地址赋值伪指令 DATA

数据地址赋值伪指令 DATA 是将数据地址或代码地址赋予规定的字符名称。其使用格式如下：

字符名称 DATA 表达式

DATA 伪指令的功能与 EQU 有些相似，使用时要注意它们有以下区别。

- EQU 伪指令必须先定义后使用，而 DATA 伪指令则无此限制；
- EQU 伪指令可以把一个汇编符号赋给一个字符名称，而 DATA 伪指令则不能；
- DATA 伪指令可将一个表达式的值赋给一个字符变量，所定义的字符变量也可以出现在表达式中，而 EQU 定义的字符则不能这样使用。DATA 伪指令在程序中常用来定义数据地址。

4.2.5 定义字节伪指令 DB

定义字节伪指令 DB 是从标号指定的地址单元开始，将数据表中的字节数据按顺序依次保存。其使用格式如下：

[标号:] DB 8 位字节数据表

这里的数据表可以是一个或多个字节数据、字符串或表达式，各项数据用“,”分隔，一个数据项占一个存储单元。定义字节伪指令 DB 的使用示例代码如下：

```
ORG 1000H
DATA1: DB 11H, 34H, 60, 90H
DATA2: DB 111B, 'A'
```

以上伪指令经汇编以后，将对从 1000H 开始的若干内存单元赋值，其中的内容如表 4-1 所示。

表 4-1 内存单元内容

内存单元	内 容	内存单元	内 容
1000H	11H	1003H	90H
1001H	34H	1004H	111B=07H
1002H	60=32H	1005H	'A'=41H

在这里，十进制数 60 和二进制数 111B 均转化为十六进制来表示，用单引号括起来的字符以 ASCII 码存入，如果遇到负数，则按补码存入。

4.2.6 定义字伪指令 DW

定义字伪指令 DW 是从标号指定的地址单元开始，将数据表中的字数据按从左到右的顺序依次保存。其使用的格式如下：

[标号:] DW 16 位字数据表

DW 的功能和 DB 相类似，需要注意的是，16 位数据要占用两个单元的存储器；其中，高 8 位数据存入低地址字节，低 8 位数据存入高地址字节。定义字伪指令 DW 的使用示例

代码如下：

```
ORG      1000H
DATA1:   DW      13ABH,    4CH
```

以上伪指令经汇编以后，将对从 1000H 开始的若干内存单元赋值，其中的内容如表 4-2 所示。

表 4-2 内存单元内容

内存单元	内 容	内存单元	内 容
1000H	13H	1002H	00H
1001H	ABH	1003H	4CH

4.2.7 定义空间伪指令 DS

定义空间伪指令 DS 是从标号指定的地址单元开始，保留若干个存储单元作为备用的空间。其中，保留的数量由表达式指定。其使用格式如下：

```
[标号:]   DS      表达式
```

定义空间伪指令 DS 的使用示例代码如下：

```
ORG      1000H
BUF:     DS      07H
```

该段伪指令经汇编以后，从地址 1000H 开始保留 7 个内存单元，然后从 1007H 开始才可以进行其他操作。

！注意：DB、DW、DS 伪指令只能对程序存储器进行定义，不能对数据存储器进行操作。DB 伪指令常用来定义数据，DW 伪指令常用来定义地址。

4.2.8 位地址符号伪指令 BIT

位地址符号伪指令 BIT，主要用于对位地址赋予所规定的字符名称。其使用格式如下：

```
字符名称  BIT      位地址
```

位地址符号伪指令 BIT 的使用示例代码如下：

```
P20      BIT      P2.0
P21      BIT      P2.1
```

该段伪指令经汇编以后，将位地址 P2.0 和 P2.1 分别赋给 P20 和 P21，此后可以将 P20 和 P21 当做位地址来代替 P2.0 和 P2.1。

4.3 汇编语言程序的格式

汇编语言程序的每一句程序一般由 4 部分组成，即标号、操作码、操作数和注释。每个部分之间要用分隔符隔开，分隔符可以采用空格、冒号“:”、分号“;”。其使用格式如下：

```
标号：      操作码    操作数    ;注释
```

一般对于一行汇编程序来说，只有操作码是必不可少的，其他视情况而定，可有可无。

标号由 8 个或 8 个以下的字符或数字构成，但第一个必须是字母。除字母和数字外，在标号中还可以使用下画线符号“_”。各种特殊功能寄存器名、各个位地址记忆符、各种伪指令等都不能用做标号。

以下是一些合法的标号。

P10、DELY、DATA0 等。

以下的字符串不能用做标号。

51 单片机开发与应用技术详解

7P、P+A、MOV 等。

操作数一般为立即数、寄存器、直接地址，以及寄存器间址等。立即数的前缀为“#”，可以是二进制，后缀为“B”，可以是十进制，没有后缀；也可以是十六进制数，后缀为“H”。十六进制数的最高位必须是数字，不能为字母，如果高位大于 9 时，则在前加 0，例如 0ABH。对直接地址 direct 来说，也有如下多种选择。

- 二进制数、十进制数或十六进制数；
- 标号地址；
- 带有加减的表达式；
- 特殊功能寄存器值。

4.4 源程序的汇编

源程序的汇编是将汇编语言源程序转换为用机器码表示的目标程序的过程。其中能完成该转换功能的程序称为汇编程序。

汇编常用的方法有两种，一种是手动汇编，另一种是机器汇编。

- 手动汇编是把程序用助记符指令写出后，然后人为查找指令代码表，逐个把助记符指令翻译成机器码，然后把得到的机器码程序（以十六进制形式）输入到单片机开发软件中，并进行调试。由于手动汇编是按绝对地址进行定位的，所以，对于偏移量的计算和程序的修改有诸多不便；而且手动汇编十分复杂，很费时间，也很容易出错，因此，一般只有程序较小或工具所限时才使用。
- 机器汇编是在计算机上，使用汇编程序将汇编语言源程序转换为计算机能识别的机器码表示的目标程序。汇编工作由计算机自动完成，生成的目标程序经调试无误后，再固化到单片机的程序存储器中。

机器汇编与手动汇编相比具有极大的优势，是汇编工作的首选。

4.5 Keil μVision3 中运行汇编语言实例

下面首先通过一个简单的实例，来介绍如何使用 μ Vision3 进行单片机汇编程序的开发。

4.5.1 创建项目

首先，启动 μ Vision3 集成开发环境，开始创建项目，操作步骤如下：

（1）选择“Project”→“New”→“μ Vision Project”命令，弹出“Create New Project”对话框，如图 4-2 所示。选择需要保存的目录并输入项目的名称，例如 Test。

（2）单击“保存”按钮，此时弹出“Select Device for Target”对话框选择 CPU 类型。可以在其中选择本项目所使用的单片机型号，也可以在项目建立后修改。例如选择 Atmel 公司的单片机 AT89S52，此时在“Description”栏中会显示该 CPU 的资源情况，如图 4-3 所示。

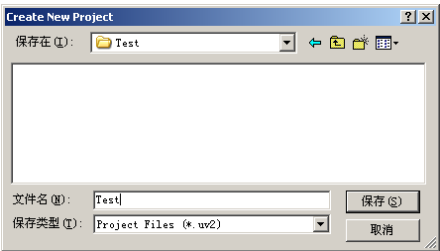


图 4-2 “Create New Project”对话框

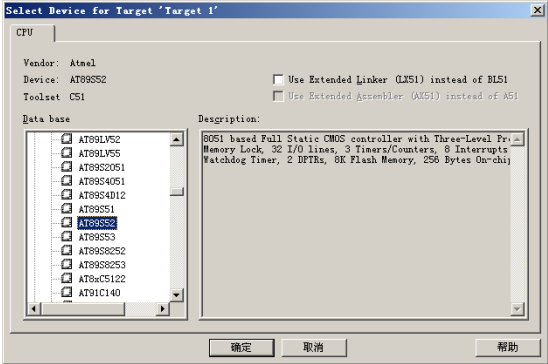


图 4-3 选择 CPU 类型

(3) 选择完毕后，单击“确定”按钮，此时弹出提示信息，如图 4-4 所示。提示是否将 8051 的起始代码添加到项目中，对于汇编程序，这里一般选择不添加。

(4) 单击“否”按钮，选择不添加，此时项目建立完毕，如图 4-5 所示。其中还没有任何源文件，属于一个空壳项目。



图 4-4 提示信息

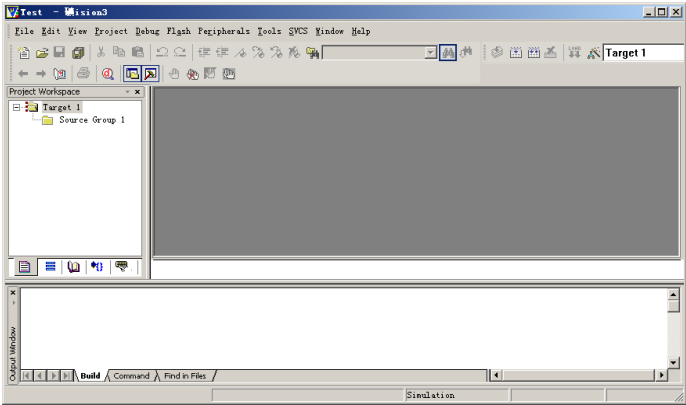


图 4-5 项目建立完毕

4.5.2 创建源文件

项目建立完毕后，现在开始进行源文件的设计，这是项目的核心。具体的操作步骤如下所述。

(1) 选择“File”→“New”命令，此时工作区中弹出一个新的文本编辑窗口，如图 4-6 所示。

(2) 可以在其中输入如下程序代码。

```
ORG      0000H
JMP      START
ORG      2000H

START:   MOV     R0,#02H    ;乘数
        MOV     R1,#05H    ;被乘数
FUMUL:   MOV     A,R1       ;取
                               ;被乘数送入累加器 A
        RLC     A           ;符号位送入 CY
        MOV     00H,C       ;存被乘数的符号
        MOV     A,R0       ;取乘数
        RLC     A           ;符号位送入 CY
```

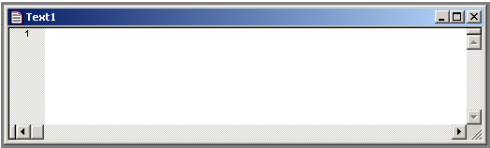


图 4-6 新建的文本编辑窗口


```
MOV    01H,C      ;存乘数的符号
ANL    C,/00H
MOV    02H,C      ;暂存到 02H 位
MOV    C,00H      ;取被乘数符号
ANL    C,/01H
ORL    C,02H      ;或运算
MOV    02H,C      ;存积的符号
MOV    A,R0       ;取乘数
JNB    ACC.7,FUN1  ;乘数为正则转向 FUN1
CPL    A          ;乘数为负则求补
INC    A
FUN1:  MOV    B,A   ;乘数存入 B
      MOV    A,R1   ;取被乘数
      JNB    ACC.7,FUN2 ;被乘数为正则转向 FUN2
      CPL    A      ;被乘数为负则求补
      INC    A
FUN2:  MUL    AB     ;相乘
      JNB    02H,FUN3 ;积为正则转向 FUN3
      CPL    A      ;积为负则求补
      ADD    A,#01H ;需用加法来加 1
FUN3:  MOV    R2,A   ;存积的低 8 位
      MOV    A,B     ;积的高 8 位送 A
      JNB    02H,FUN4 ;积为正则转向 FUN4
      CPL    A      ;高 8 位取反
      ADDC   A,#00H  ;加进位
FUN4:  MOV    R3,A   ;存积的高 8 位
      RET
      END          ;返回
```

这段代码演示了 8 位带符号整数的乘法汇编程序。被乘数和乘数分别保存在 R1 和 R0 中，程序结束时，将积的高 8 位和低 8 位分别保存在 R3 和 R2 中。

(3) 代码输入完毕后，可以单击保存按钮，将其保存为 test.asm 文件。

(4) 在项目管理窗口中，右键单击“Source Group 1”，选择“Add Files to Group ‘Source Group 1’”命令，在弹出的对话框中选择刚才保存的汇编源文件，并加入项目中即可。

4.5.3 编译项目

项目及源文件建立完毕后便可以编译项目了。选择“Project”→“Build target”命令，即可编译，如果程序无误，则在输出窗口中显示编译结果，如图 4-7 所示。

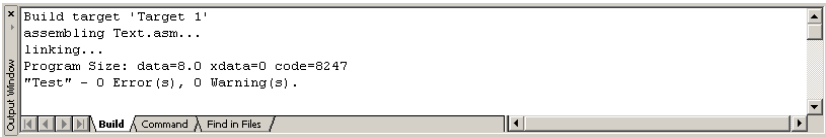


图 4-7 编译输出结果

如果需要生成单片机上可执行的文件，可以选择“Project”→“Options for Target ‘Target 1’”命令，此时弹出“Options for Target ‘Target 1’”对话框，如图 4-8 所示。在“Output”选项卡中，

选择复选框“Create Hex File”，并单击“确定”按钮保存设置。

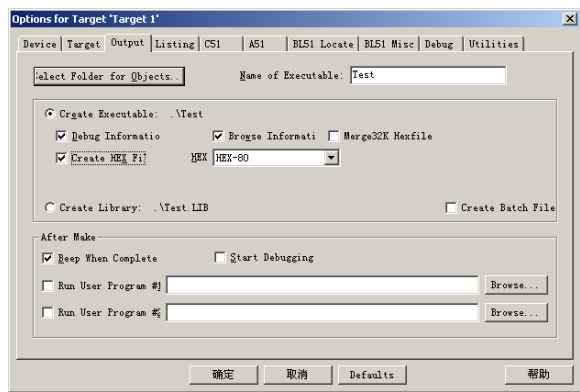


图 4-8 “Options for Target ‘Target 1’” 对话框

此时，重新编译一次，便生成可以下载到单片机中的执行文件 Test.hex；然后可以利用下载工具将其下载到单片机中执行。

4.5.4 仿真调试

项目建立并编译通过后，在下载单片机硬件之前，一般需要进行仿真调试，确保程序的执行完全符合要求。 μ Vision3 的集成开发环境提供了完善的项目仿真和调试工具，下面介绍如何使用 μ Vision3 来进行项目的仿真调试。

项目仿真调试之前需要确保该项目完全通过编译。仿真调试的步骤如下所述。

（1）选择“Debug”→“Start/Stop Debug Session”命令，此时， μ Vision3 进入仿真调试模式。界面的菜单栏和工具栏都相应地进行了变化。

（2）使用单步执行的方式运行程序。每按 F11 键一次，程序执行一条指令。用户可以看到程序执行时，单片机的各个资源和寄存器值是如何变化的。

（3）当程序执行完毕的时候，在寄存器显示窗口中可以看到程序仿真的结果，如图 4-9 所示。乘法运算的结果保存在寄存器 R3 和 R2 中。

用户可以自行修改寄存器 R0 和 R1 中的数据，重新运行来查看仿真结果。

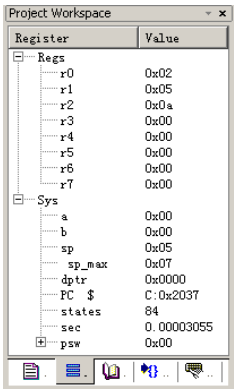


图 4-9 程序仿真结果

4.6 小结

本章主要讲述了汇编语言的伪指令、汇编语言的程序格式及源程序的编译。本章最后还通过一个具体的实例，介绍了如何在 Keil μ Vision3 集成开发环境中进行单片机的汇编程序设计及仿真。汇编语言是最早应用于单片机开发的程序语言。相比其他程序设计语言，汇编语言指令的执行速度快、代码短小精悍，且具有确定的指令执行周期。熟练掌握本章内容是后续汇编语言程序学习的基础。

第5章 汇编语言程序结构

汇编语言程序结构是汇编程序所采用的结构形式。汇编语言程序共有 4 种结构形式，即顺序结构、分支结构、循环结构和子程序结构。用汇编语言进行程序设计，与用高级语言（如 C 语言等）进行程序设计的过程很相似。对于比较复杂的问题可以先根据设计的要求，选用不同的程序结构，然后做出流程图，最后再根据流程图来编写程序。对于比较简单的问题则可以不画流程图而直接编程。这里除了讲解这 4 种基本的结构形式外，还专门讨论了常用的查表结构和运算类程序的设计。

5.1 顺序结构程序

顺序结构程序是一种无分支的直线型程序结构，即按照程序编写的顺序依次执行每一条指令。它是一种最简单、最基本的程序，所以有时也称为简单程序结构。下面举两个顺序结构的程序设计实例，其中用到的汇编指令将在后面的指令系统中详细介绍。

1. 实例 1

首先是一个 16 位二进制数求补的汇编程序。这里假定带操作的这个双字节数存放在 R3、R4 中，程序将求补以后的结果存放于地址 20H、21H 中。

这个程序的基本思路是，首先低字节求补，即“求反加一”；然后，进行高字节的求补，此时，应该注意低字节加 1 时可能产生的进位。采用顺序结构的程序示例如下：

```
ORG      0200H          ; 汇编程序起始命令
MOV      A, R4           ; 低字节送入累加器 A
CPL      A               ; 取反
ADD      A, #01H         ; 加 1
MOV      21H, A          ; 将结果送入地址 21H
MOV      A, R3           ; 高字节送入累加器 A
CPL      A               ; 取反
ADDC A, #00H             ; 加进位位
MOV      20H, A          ; 将结果送入地址 20H
END                ; 程序结束
```

2. 实例 2

再举一个计算两个 16 位二进制无符号数加法的程序例子。16 位二进制无符号数在内存中占有两个单元，这里假定已经分别存于寄存器 R1、R0，R3、R2 中。其中，R1 和 R3 分别为高字节，R0 和 R2 分别为低字节。两个数相加后的和存放在 30H、31H、32H 单元中。采用顺序结构的程序示例如下：

```
ORG      0200H          ; 汇编程序起始命令
```

```
CLR      C                                ; 清除进位标志
MOV      A,R0                            ; 将一个加数的低字节送到累加器 A 中
ADD      A,R2                            ; 累加器 A 中的数据 and 另一个加数相加，并将结果送入 A
MOV      32H,A                          ; 将结果送入内存单元 32H 中
MOV      A,R1                            ; 将一个加数的高字节送到累加器 A 中
ADDC     A,R3                            ; A 和另一个加数的高字节进行带进位相加
MOV      31H,A                          ; 将结果送入内存单元 31H 中
MOV      A,#00H                         ; 加进位位
ADDC     A,#00H                         ; 加进位位
MOV      30H,A                          ; 将和数的进位位送入内存单元 30H 中
END                                          ; 程序结束
```

以上两例都是简单的顺序程序，可以完成一些简单的功能，如果程序的末尾不用 `END` 结束，而用返回指令 `RET` 结束，则可以当做完成某些特定功能的程序段，这相当于子程序。

5.2 分支结构程序

分支结构程序是根据判断条件的满足与否，产生一个或多个程序分支，以实现不同的程序流向的程序结构。在一些实际的应用程序中，程序不可能始终是按顺序直线执行的。要使用单片机解决一些实际的问题，通常要求单片机能够做出一些判断，从而实现分支结构程序。

分支程序可以分为双分支结构和多分支结构两种，分别如图 5-1 和图 5-2 所示。下面分别介绍这两种分支结构的程序设计。

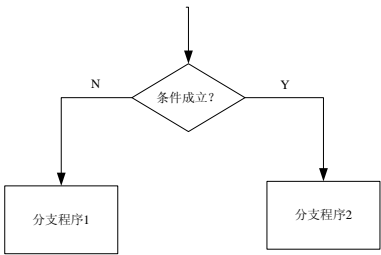


图 5-1 双分支结构

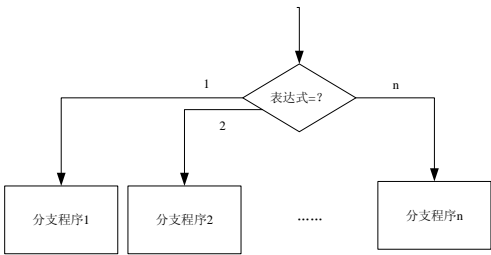


图 5-2 多分支结构

5.2.1 双分支结构

双分支结构主要采用条件转移指令来实现分支转移，当给定的条件成立时，执行分支程序 1，否则执行分支程序 2。

编写分支程序的关键是进行分支条件的判断。在 51 系列单片机中，主要有位条件转移指令 `JC`、`JB` 等，比较条件转移指令 `CJNE` 等和累加器 `A` 判断指令 `JZ` 等，这些指令的详细介绍参阅本书第 16 章指令系统部分。合理使用这些指令可以完成各种各样的条件判断。

下面仍以前面的 16 位二进制数求补的汇编程序为例。这个程序也可以用分支结构的程序设计，这里仍假定带操作的这个双字节数存放在 `R3`、`R4` 中，程序将求补以后的结果存放于地址 `20H`、`21H` 中。程序的流程图，如图 5-3 所示。程序示例如下：

```
ORG      0200H                          ; 汇编程序起始命令
MOV      A,R4                            ; 低字节送入累加器 A
CPL      A                               ; 取反
ADD      A,#01H                          ; 加 1
```

```
MOV      21H,A      ;将结果送入地址 21H
JZ       ZERO       ;如果 A 的值为零则转向 ZERO
MOV      A,R3        ;高字节送入累加器 A
CPL      A           ;取反
MOV      20H,A       ;将结果送入地址 20H
SJMP     FEND        ;转结束

ZERO:    MOV      A,R3        ;高字节送入累加器 A
CPL      A           ;取反
INC      A           ;加一
MOV      20H,A       ;将结果送入地址 20H

FEND:    END          ;程序结束
```

在该段程序中，首先对低字节数取补，然后判断其结果是否为 0，如果为 0，则对高字节数进行取补，即取反加 1，否则直接取反就可以了。这个算法和前面的算法，结果是一样的。

有些简单的多分支结构的程序，也可以用双分支来分步实现。例如符号函数 $Y=\text{sign}(X)$ 。当 $X>0$ 时， $Y=1$ ；当 $X=0$ 时， $Y=0$ ；当 $X<0$ 时， $Y=-1$ 。下面采用汇编语言来实现这个函数，假定变量 X 已经存放在 $20H$ 单元，将函数值 Y 存放于 $21H$ 单元。程序的流程如图 5-4 所示。

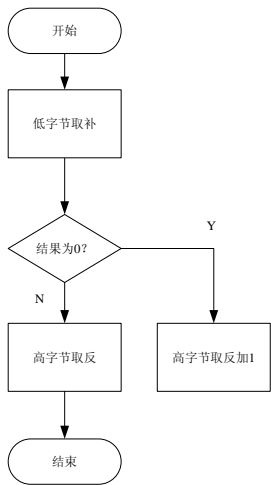


图 5-3 16 位二进制数求补流程图

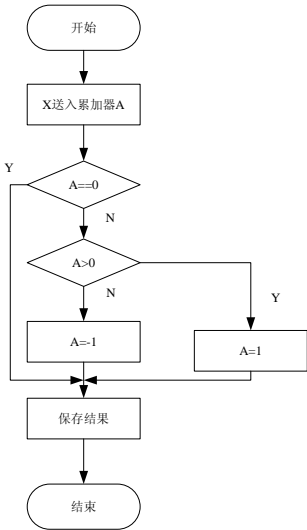


图 5-4 符号函数流程图

程序示例代码如下：

```
ORG      0200H      ;汇编程序起始命令
MOV      A,20H       ;将 X 的值送入累加器 A
JZ       F1          ;如果 X=0，则转向 F1 处理
JNB      ACC.7,F2     ;如果 X>0，则转向 F2 处理
MOV      A,#0FFH     ;如果 X<0，则 Y=-1
SJMP     F1

F2:      MOV      A,#1 ;如果 X>0，则 Y=1
F1:      MOV      21H,A ;保存 Y 的值
END
```

该程序中，相当于分别运用了两个双分支结构来实现运算。

5.2.2 多分支结构

多分支结构是根据运算的结果指在多个分支中选择一个执行的程序结构。双分支是比较简单的情况，在实际的应用中，往往需要多分支跳转，又称为散转。51 系列单片机的指令集中有散转指令 **JMP**，详细的介绍可以参阅后面的指令系统介绍。散转指令 **JMP** 的使用格式如下：

```
JMP      @A+DPTR
```

其中数据指针 **DPTR** 为存放转移指令串 (**S0~Sn**) 的首地址，由累加器 **A** 的内容动态选择对应的转移指令。这样，便可以产生多达 256 个分支程序。

51 系列单片机的指令系统中，有两个无条件转移指令，即 **AJMP** 和 **LJMP**。前者为双字节指令，后者为三字节指令。这将直接影响散转指令中对累加器 **A** 的处理。即如果选择 **AJMP**，则应该将累加器 **A** 值变换成偶数值；如果选择 **LJMP** 指令，则应该对累加器 **A** 值进行乘 3 的变换。

汇编语言程序中，每个分支程序都有各自的程序段，分别位于程序存储器的不同区段。判断语句根据判断结果，使程序转向不同的分支程序段，这个无条件转移指令串的首地址由 **DPTR** 指示。这里以 **AJMP** 为例，介绍散转指令的应用。示例代码如下：

```
MOV      DPTR, #ADDR
CLR      C
RLC      A
JNC      SEL
INC      DPH
SEL:     JMP      @A+DPTR
.....
ADDR:    AJMP F0
          AJMP F1
          .....
          AJMP Fn
F0:      .....
          .....
F1:      .....
          .....
```

在该段程序中，由于 **AJMP** 为双字节绝对转移指令，因此，要求累加器 **A** 中的内容必须换算成偶数。换算方式可以乘 2，即相当于左移一位。如果 **n** 的值等于或大于 128，则左移一位将产生高位进位，将进位值加到 **DPH** 中，这相当于将转移指令串的首地址延伸 256 个存储单元，所以在程序中对 **C** 进行测判。这样可以保证分支程序可在 0~255 个中任意选择。

如果汇编程序中使用了三字节的长调用 **LJMP** 指令，在进行累加器 **A** 值换算时应该乘 3，将积的高字节值加到 **DPH** 中去。

！注意：一般 **DPTR+A** 的最终值应该不超过 64KB。

5.3 循环结构程序

循环结构程序是重复执行同一个程序段的一种基本程序结构。实际应用中，经常会遇到需要多次执行某段特定代码的情况，这时可以采用循环程序，缩短程序的长度，节省程序的存储空间。从本质上来说，循环结构程序是分支结构程序的一种特殊形式。由于它在程序设计中的重要性，因此一般单独作为一种程序结构的形式来介绍。

5.3.1 循环程序的结构和组成

一个典型的循环程序由 4 部分组成，即循环初始化部分、循环处理部分、循环控制部分和循环结束部分。下面分别进行介绍。

- 循环初始化部分：程序在进入循环处理程序段之前，需要设置循环初始参数，如循环的次数、有关的工作单元清零、变量设置和地址指针设置等。
- 循环处理部分：循环处理部分通常称为循环体，是循环执行的主要代码段，它是整个循环结构的核心。
- 循环控制部分：循环控制部分一般由两个单元组成，修改控制变量和判断循环结束。循环控制变量可以采用循环递减计数法，即每循环一次，控制变量减 1，并判断是否为 0，若不为 0，则继续执行循环体程序，否则结束循环体的执行；也可以采用条件控制，即判断结束条件是否成立，如果不成立，则继续执行循环体，否则，结束循环。
- 循环结束部分：当循环体执行完毕后，需要在这里对结果进行处理和存储。

循环结构大致可以分为两种，分别如图 5-5 和图 5-6 所示。第一种是先执行后判断的循环结构，适用于循环次数已知的情况。其特点是一进入循环，先执行循环处理部分，然后根据循环次数或循环条件来判断是否继续循环。第二种是先判断后执行的循环结构，适用于循环次数未知的情况。其特点是将循环的控制部分放在循环的入口处，这样，进入循环体时，需要先根据循环控制条件来判断是否结束循环，如果不结束循环，则执行循环体，否则，则退出循环。

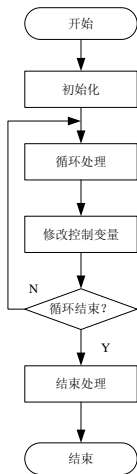


图 5-5 先执行后判断的循环结构

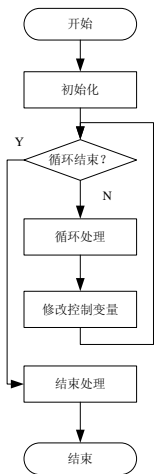


图 5-6 先判断后执行的循环结构

5.3.2 循环程序示例

51 系列单片机提供了两条循环转移指令，示例如下：

```
DJNZ Rn, LOOP      ;采用工作寄存器 Rn 为控制寄存器
DJNZ Direct, LOOP   ;采用直接寻址单元 Direct 作为控制寄存器
```

这两条指令采用不同的控制寄存器来实现循环结构，关于它们的详细介绍可以参阅本书第 16 章中的指令系统部分。这里，控制寄存器的计数方式一般都是减 1 计数，即每循环一次，计数器自动减 1 计数，同时判断控制寄存器是否为 0，若不为 0，继续执行循环；若为 0，则结束循环程序的执行。循环次数需要在初始化的时候预置，循环次数的范围为 1~255。如果实际问题中需要超过 255 个循环的时候，则可以采用多重循环来实现。

下面给出一个采用循环结构来查找最大值的程序。假定从 20H 单元开始连续存放了 20 个无符号整数，用汇编程序找出其中的最大值，并存入 R2 中。

寻找最大值的方法比较多，一个最基本的方法是采用比较和交换来依次进行。即先读取第一个数据与第二个数据，把第一个数据作为基准，进行比较。如果基准数大，则不做交换，再

取下一个数进行比较；如果基准数小，则将大数取代原来的基准数，即相当于做一次数据交换，然后再以新的基准数与下一个数进行比较，直至全部比较完毕。这里的基准数始终保持为最大的数值，因此，全部比较完毕后，基准数即是数据块中的最大值。整个程序的流程图，如图 5-7 所示。采用循环结构的程序示例如下：

```
ORG      0400H
MOV      R0, #20H      ;数据块首地址送入 R0
MOV      A, @R0        ;取第一个数作为基准数送入累加器 A
MOV      R1, #13H      ;比较次数 19=13H 送入 R1
LOOP1:   INC      R0    ;修改地址指针，使其指向下一个地址单元
MOV      20H, @R0      ;将要比较的数暂存于 20H 单元
CJNE A, 20H, CHK        ;两个数进行比较
CHK:     JNC      LOOP2  ;如果 A 大，则转换
MOV      A, @R0        ;如果 A 小，则将较大的数送入 A
LOOP2:   DJNZ R1, LOOP1 ;R1 减 1，如果其不为 0，则继续循环
MOV      R2, A         ;比较完毕，存结果
END
```

另外，延时在程序中经常用到，延时可以用中断、计数和循环等多种方法来实现。采用循环程序来实现延时，一般可以达到任意延时的要求。采用循环延时的程序代码示例如下：

```
ORG      1000H
MOV      20H, #DATA    ;设置计数初始值 DATA
LOOP:    NOP            ;空指令 NOP
NOP
NOP
DJNZ 20H, LOOP          ;判断 ( 20H ) -1 是否为 0，如果不为 0 则继续循环
END
```

在该段程序中，主循环中采用空指令 NOP 来实现延时，延时的大小根据其中 NOP 指令的多少和每个指令的执行周期及单片机系统的主频决定。程序的流程图，如图 5-8 所示。

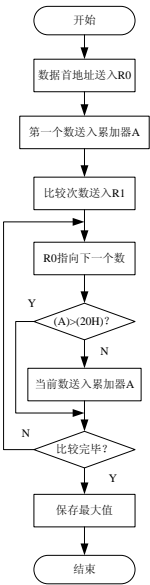


图 5-7 最大值查找流程图

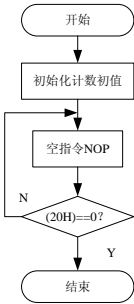


图 5-8 循环延时流程图

51 单片机开发与应用技术详解

采用循环延时的方法，虽然可以达到任意延时的要求，但其也有缺点。比如需要牺牲 CPU 的工作，延时期间，CPU 不能做任何事情，同时循环延时的方法一般不允许程序中有中断，否则会严重影响延时的准确性。

5.3.3 多重循环程序结构

多重循环程序结构是循环体多于一个的程序结构。实际的程序设计中，经常遇到循环次数多于 256 的情况，这时必须采用多重循环来实现，既一个循环中又包含了一个或多个循环，因此也称为循环嵌套。多重循环的嵌套次数没有限制，这类循环主要应注意循环的层次要清晰，不要产生交叉，否则会造成系统紊乱。

以双重循环为例，前面的循环延时如果采用双重循环，则可以实现大的软件延时。示例如下：

```

                                ORG      0400H
                                MOV      20H, #DATA1    ;设置计数初始值 DATA1
                                MOV      21H, #DATA2    ;设置计数初始值 DATA2
LOOP1:                          MOV      NOP          ;空指令 NOP
LOOP2:                          NOP
                                NOP
                                DJNZ 21H, LOOP2        ;判断 ( 21H ) -1 是否为 0，如果不为 0 则继续循环 LOOP2
                                DJNZ 20H, LOOP1        ;判断 ( 20H ) -1 是否为 0，如果不为 0 则继续循环 LOOP1
                                END
```

5.4 子程序结构

子程序结构是将某些运算和操作设计成一小段可被其他程序调用的程序段，需要的时候直接调用这些程序段的程序结构。其中能够完成特定功能、可以被其他程序调用的程序段称为子程序。调用子程序的程序称为主程序，调用子程序的过程称为子程序调用。子程序执行完后返回主程序的过程称为子程序返回。

在实际的程序设计中，经常会遇到一些相同的操作，如多字节加法、代码处理等。此时，采用子程序结构将会省去很多重复编写程序段的麻烦，而且可以缩短程序代码，使程序紧凑，结构清晰明了。

5.4.1 子程序的结构

子程序是具有特定功能的独立程序段，子程序的结构需要具备如下特点。

- 子程序必须提供入口地址，以便于主程序调用；
- 子程序必须以返回指令 **RET** 结束子程序。
- 在汇编主程序中调用子程序时，需要注意参数传递和现场保护两个问题。参数传递需要用户自己安排，在主程序中，调用子程序的指令不带任何参数。汇编程序通常采用下面两种方法来进行参数传递。
- 传递数据。在主程序调用子程序前，将需要传递的参数送入通过工作寄存器 **R0~R7** 或累加器 **A**，供子程序读取使用。
- 传递地址。主程序将要传递的参数存放在数据存储器中，其地址送入工作寄存器 **R0~R7** 或数据指针 **DPTR**，供子程序读取使用。

现场保护是指在调用子程序前，将这些寄存器和累加器中的内容保护起来。现场保护是一个很重要的问题。在主程序调用子程序之前，累加器 **A** 或工作寄存器中可能存放着主程序运算的中间结果，这些中间结果在主程序中仍然有用，而子程序中也要用到这些寄存器或累加器。这时就需要现场保护，否则将会使前面的寄存器内容丢失。当子程序执行完毕后，将这些数据取出，送到原来的寄存器或累加器中，这个过程称为恢复现场。

现场保护一般采用堆栈来操作。在需要保护现场的时候，用压栈指令 **PUSH**，将需要保护的寄存器内容压入堆栈。在子程序执行完后，在返回指令 **RET** 前使用弹栈指令 **POP**，把堆栈中保护的内容恢复到原来的寄存器中。

！注意：在汇编程序中，堆栈操作是“先入后出”的，因此，先压入堆栈的参数应该后恢复，才能保证将数据恢复到原来的寄存器。

5.4.2 子程序的调用与返回

51 系列单片机指令集中提供了两个指令可以用来调用子程序，其使用格式如下：

```
LCALL ADDR16
```

以及如下：

```
ACALL ADDR11
```

LCALL 称为长调用指令，指令的操作数 **ADDR16** 给出了子程序的 16 位入口地址；**ACALL** 称为绝对调用指令，其中的操作数 **ADDR11** 提供了子程序的低 11 位入口地址，这个地址和程序计数器 **PC** 的高 5 位并在一起，便构成了 16 位的调用地址，即子程序的入口地址。

在执行时，子程序调用指令首先将 **PC** 中的内容（调用指令下一条指令地址，称为断点地址）压入堆栈，即断点保护，然后将调用地址送入 **PC** 中，使程序跳转到子程序的入口地址。

返回指令 **RET**，用于子程序的返回。该指令将堆栈中存放的返回地址（即断点地址）弹出堆栈，送回到 **PC** 中，使程序返回到主程序的断点处继续向下执行。

5.4.3 子程序设计实例

下面首先举一个简单的例子，用汇编语言计算 $Y=a \times a + b \times b$ 。

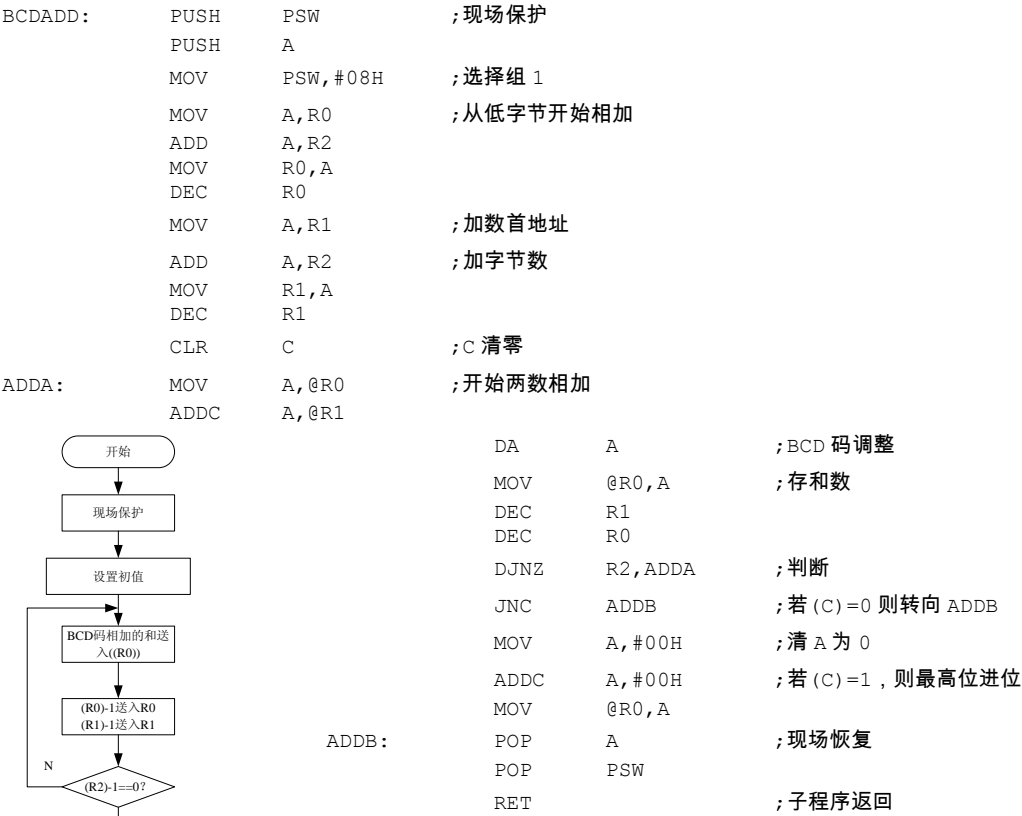
首先假定 **a** 存放在 20H 单元，**b** 存放在 21H 单元，计算的结果存放在 22H 单元中。其中的 **a** 和 **b** 是小于 10 的无符号整数。这里 $a \times a$ 的运算可以采用子程序的方法，在子程序中，通过查表来求得平方值。在调用子程序时，将 **a** 送入累加器 **A** 中，然后在子程序结束的时候，将 $a \times a$ 保存在累加器 **A** 中，从而实现参数的传递。主程序示例如下：

```
ORG 2000H
MOV SP, #3FH ; 设堆栈指针，调用和返回指令时用到
MOV A, 20H ; 取 A 的数值
LCALL FUNC ; 调用子程序，求 a*a
MOV R1, A ; 计算的结果存入 R1
MOV A, 21H ; 取 b 的数值
LCALL FUNC ; 调用子程序，求 b*b
ADD A, R1 ; 计算 a*a+b*b
MOV 22H, A ; 保存结果
SJMP $ ; 暂停
```

子程序示例如下：

```
ORG 0200H
FUNC: ADD A, #01H ; 查表位置
      MOVC A, @A+PC ; 查表取平方
      RET ; 子程序返回
TAB: DB 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ; 平方数值表
```

接着，再举一个多字节 **BCD** 码加法的子程序。程序代码示例如下：



```

                                ORG      1000H
HEX1:    EQU      20H          ;赋值伪指令
HEX2:    EQU      21H
HTOAS:   MOV      A,HEX1
                                ANL      A,#00001111B
                                ADD      A,#03H          ;变址调整
                                MOVC     A,@A+PC
                                MOV      HEX2,A
                                RET
DATA:    DB      30H,31H,32H,33H  ;ASCII 码表格
                                DB      34H,35H,36H,37H
                                DB      38H,39H,40H,41H
                                DB      42H,43H,44H,45H
                                END
```

在本程序中，查表指令 **MOVC A,@A+PC** 到表格首地址 **DATA** 有两个指令，占用三个地址空间，因此程序中采用了变址调整为 3。

这里再举一个无序表格的例子。从 10 个人的档案中，查找一个人 **ZHAO** 的名字，此人的名字存于 **20H** 单元。如果找到，则记录其地址并存入寄存器 **R6、R5** 中，否则，将 **R6、R5** 清零。档案表格的首地址为 **TAB**。

这显然是一个无序的表格，无法用算法来实现，只能逐个单元查找。采用查表结构的程序代码示例如下：

```

                                ORG      1000H
ZHAO:    EQU      20H
FZHAO:   MOV      B,ZHAO        ;关键字送入 B
                                MOV      R4,#10          ;查找次数送入 R4
                                MOV      DPTR,#TAB
                                MOV      A,#15H          ;变址调整
LOOP:    PUSH     A              ;暂存 A
                                MOVC     A,@A+PC          ;查表
                                CJNZ     A,B,NF            ;如果没有找到则转 NF
                                MOV      R6,DPH            ;找到并记录地址
                                MOV      R5,DPL
                                POP      A
DONE:    RET
NF:      POP      A              ;恢复 A
                                INC      A                ;求下一个地址
                                INC      DPTR              ;数据表地址加 1
                                DJNZ     R4,LOOP           ;未完则继续
                                MOV      R6,#00H
                                MOV      R5,#00H          ;如果没有找到，则 R6、R5 清零
                                AJMP     DOWN
TAB:     DB      'A'          ;数据表示例
                                DB      'B'
                                DB      'C'
                                DB      'D'
                                DB      'E'
                                DB      'F'
                                DB      'G'
                                DB      'H'
                                DB      'I'
                                DB      'J'
                                END
```

在这段示例程序中，数据表只简单地用字符来表示，也可以根据需要更改。另外增加数据

表的量，也可以实现比较大的数据表的查询。

5.6 运算类程序

运算类程序是指专门负责算术或逻辑运算的程序。在实际的程序设计中，经常会遇到一些数学计算类的程序，比如多字节数的加法、减法、乘法和除法，以及带符号数的运算等。由于 51 系列单片机的指令系统中，只提供了单字节和无符号数的算术运算指令，因此需要自己编写这些子程序，这样可以方便以后使用。下面给出一些典型的运算类子程序，以方便读者学习使用。

5.6.1 8 位带符号整数的乘法子程序

实际的程序设计中，如果遇到需要处理带符号数的乘法，则需要编写带符号整数的乘法的子程序，因为 51 系列单片机中的乘法指令只适用于无符号数。

带符号数乘法需要完成的工作是：首先保存两个乘数的符号，求出整个乘积结果的符号；然后，两个数取绝对值进行相乘；最后，根据积的符号，对得到的结果进行处理。

本子程序中，参数的传递是这样的：被乘数和乘数分别保存在 R1 和 R0 中，供子程序使用；子程序结束时，将积的高 8 位和低 8 位分别保存在 R3 和 R2 中。8 位带符号整数的乘法子程序代码示例如下：

```
FUMUL:      MOV      A,R1          ;取被乘数送入累加器 A
             RLC      A            ;符号位送入 CY
             MOV      00H,C        ;存被乘数的符号
             MOV      A,R0          ;取乘数
             RLC      A            ;符号位送入 CY
             MOV      01H,C        ;存乘数的符号
             ANL      C,/00H
             MOV      02H,C        ;暂存到 02H 位
             ANL      C,00H        ;取被乘数符号
             MOV      C,/01H
             ORL      C,02H        ;或运算
             MOV      02H,C        ;存积的符号
             MOV      A,R0          ;取乘数
             JNB      ACC.7,FUN1    ;乘数为正则转向 FUN1
             CPL      A            ;乘数为负则求补
             INC      A
FUN1:        MOV      B,A          ;乘数存入 B
             MOV      A,R1          ;取被乘数
             JNB      ACC.7,FUN2    ;被乘数为正则转向 FUN2
             CPL      A            ;被乘数为负则求补
             INC      A
FUN2:        MUL      AB           ;相乘
             JNB      02H,FUN3      ;积为正则转向 FUN3
             CPL      A            ;积为负则求补
             ADD      A,#01H        ;需用加法来加 1
FUN3:        MOV      R2,A          ;存积的低 8 位
```

```
MOV      A,B           ;积的高 8 位送 A
JNB      02H,FUN4      ;积为正则转向 FUN4
CPL      A             ;高 8 位取反
ADDC     A,#00H        ;加进位
FUN4:    MOV      R3,A  ;存积的高 8 位
RET                      ;返回
```

在该程序中，积的符号使用位运算指令进行异或操作，即通过位的与、或运算来完成。正数的绝对值是其原码本身，负数的绝对值是通过求补码来实现的。本程序的流程图，如图 5-10 所示。

5.6.2 8 位带符号整数的除法子程序

同前面类似，如果遇到需要处理带符号数的除法，则需要编写带符号整数的除法的子程序，来克服指令集中的除法指令的限制，因为 51 系列单片机中的除法指令只适用于无符号数。8 位带符号整数的除法子程序采用的方法是：将被除数和除数的绝对值进行相除，根据两数的符号来决定最后商的符号，另外，余数的符号要和被除数相同。

本子程序的参数传递情况是：被除数和除数分别保存在 R1 和 R2 中，供子程序使用，子程序执行完毕后，将商和余数分别保存在 R1 和 R2 中，即覆盖了原来的数据。8 位带符号整数的除法子程序的流程图，如图 5-11 所示。程序代码示例如下：

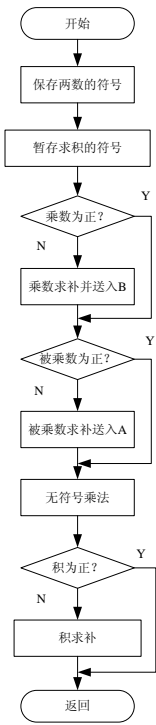


图 5-10 8 位带符号整数乘法流程图

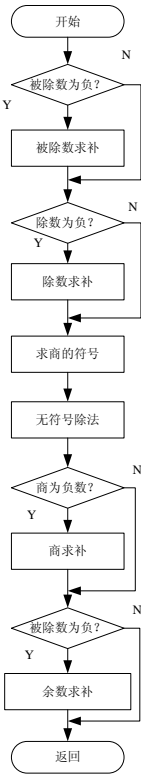


图 5-11 8 位带符号整数除法流程图

```
FUDIV:    MOV      A,R1      ;开始求被除数的符号
          ANL      A,#80H
          MOV      R4,A      ;存被除数的符号
```

```

        JZ      FUN2      ;正数则转向 FUN2
FUN1:    MOV     A,R1      ;被除数求补
        CPL     A
        INC     A
        MOV     R1,A
        FUN2:    MOV     A,R2      ;开始求除数符号
        ANL     A,#80H
        MOV     R5,A      ;存除数符号
        JZ      FDIV      ;正数则转向 FDIV
        MOV     A,R2      ;除数求补
        CPL     A
        INC     A
        MOV     R2,A
        FDIV:    MOV     A,R4      ;求商的符号
        XRL     A,R5
        MOV     R5,A      ;存符号
        MOV     A,R1      ;求商
        MOV     B,R2
        DIV     AB
        MOV     R1,A      ;存商
        MOV     R2,B      ;存余数
        MOV     A,R5      ;取商的符号
        JZ      FUN4      ;商为正则转向 FUN4
        FUN3:    MOV     A,R1      ;商为负则求补
        CPL     A
        INC     A
        MOV     R1,A
        FUN4:    MOV     A,R4      ;取被除数符号
        JZ      FRET      ;为正则转向 FRET，即返回
        MOV     A,R2      ;为负，则余数求补
        CPL     A
        INC     A
        MOV     R2,A
        FRET:    RET
```

在该程序中，使用了工作寄存器 R4 暂存被除数的符号，R5 暂存除数的符号或商的符号。这里确定商的符号是采用字节的与运算、异或运算来确定商的符号，当然也可以采用上一个例子的方法来实现。

5.7 小结

本章主要介绍了利用汇编语言进行单片机设计的各种程序结构，包括顺序结构、分支结构、循环结构和子程序结构。本章还介绍了两类比较常用的程序设计类型，即查表程序和运算类的程序。对于比较复杂的问题可以根据设计的要求，选用不同的程序结构来进行设计。因此，熟练掌握本章内容，对读者以后的设计工作会很有帮助。

第三篇 编程篇——C51

第6章 单片机 C 语言程序设计基础

前面一章介绍了单片机的汇编语言程序。在单片机的开发与应用中，除了汇编语言，也可以采用高级语言，如 C 语言。单片机 C 语言既有汇编语言操作底层硬件的能力，又具有高级语言的许多优点。因此，在现代的单片机程序设计中广泛采用单片机 C 语言。本章主要介绍了单片机 C 语言程序设计的基础知识，包括标识符、关键字、数据类型、表达式和运算符等，熟练掌握这些知识，将为后面学习 C51 的函数和语句打好基础。

6.1 单片机 C 语言概述

对比现有的程序设计语言，C 语言以其功能强大、结构清晰等优点获得广泛应用。对于学过 C 语言的读者来说，再学其他语言就显得轻而易举了。而在单片机程序设计中，同样有应用于单片机的 C 语言，通常简称为 C51 语言。C51 语言继承了 C 语言结构上的优点，便于学习，又有汇编语言操作硬件的能力，因此被广泛使用于单片机程序设计中。

6.1.1 单片机 C 语言和汇编语言对比

汇编语言是一种面向机器的程序语言，其可以直接控制硬件，指令执行速度快，且指令的执行时间固定，因此汇编语言的执行效率很高。但其语言格式比较晦涩、可读性差、难于编写和调试，也不便于移植，影响了程序代码的共享。

与汇编语言相比，单片机 C 语言在结构上更易理解、可读性强，且开发速度快、可靠性好、便于移植。因此，使用单片机 C 语言进行单片机系统的开发，可以缩短开发周期、降低开发成本。随着单片机硬件系统的发展和产品更新速度的提升，单片机的开发越来越侧重于程序本身的开发效率，以便快速占领市场。因此，单片机 C 语言已经成为目前最流行的单片机开发语言。

6.1.2 C51 语言的主要特点

单片机 C51 语言特点很多，总结起来主要的有以下几点。

- 单片机 C51 语言兼备高级语言与低级语言的优点，语法结构和标准 C 语言基本一致。其规模适中，语言简洁，便于学习。
- 同 C 语言一样，单片机 C51 语言提供了完备的数据类型、运算符及函数供使用。
- C51 语言借鉴了 ALGOL 的分程序结构，是一种结构化程序设计语言。其使用一对花括号“{}”将一系列语句组合成一个复合语句，程序结构简单明了。
- C51 语言的可移植性好。对于兼容的 8051 系列单片机，只要将一个硬件型号下的程序稍加修改，甚至不加改变，就可移植到另一个不同的硬件型号开发环境中使用。
- C51 语言生成的代码执行效率高，且比汇编语言的程序易于理解和共享使用。

随着技术的发展，C51 语言在代码执行的效率方面越来越接近汇编语言。另外，单片机的程序存储器逐渐扩大容量，硬件的工作频率也越来越快，这些都使得 C51 的应用越来越广泛。目前 C51 已经成为开发 8051 系列单片机最流行的工具。

6.2 单片机 C 语言在 Keil μ Vision3 中应用实例

单片机 C 语言（即 C51 语言）是运行在单片机上的程序语言，和 C 语言的语法结构是基本一致的。下面通过一个实例，来讲解如何在 Keil μ Vision3 集成开发环境下进行单片机 C 语言的程序设计。

6.2.1 创建项目

启动 Keil μ Vision3 集成开发环境，首先开始创建项目，操作步骤如下：

（1）选择“Project”→“New”→“ μ Vision Project”命令，弹出“Create New Project”对话框，如图 6-1 所示。选择需要保存的目录并输入项目的名称，例如 Test。

（2）单击“保存”按钮，此时弹出“Select Device for Target”对话框。可以在其中选择本项目所使用的单片机型号，也可以在项目建立后修改。例如选择 Atmel 公司的单片机 AT89S52，此时在“Description”栏中将会显示该 CPU 的资源情况，如图 6-2 所示。

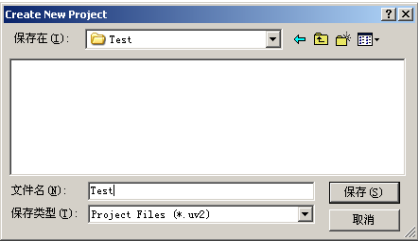


图 6-1 “Create New Project”对话框

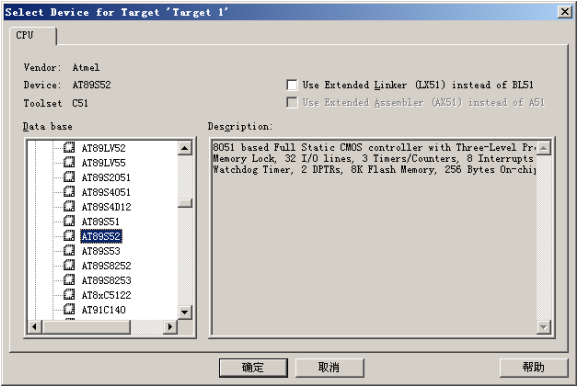


图 6-2 “Select Device for Target”对话框

（3）选择完毕后，单击“确定”按钮，此时弹出提示信息，如图 6-3 所示。提示是否将 8051 的起始代码添加到项目中，对于 C51 程序，这里一般选择添加。

（4）单击“是”按钮，选择添加，此时项目建立完毕，如图 6-4 所示。其中还没有任何源文件，属于一个空壳项目。

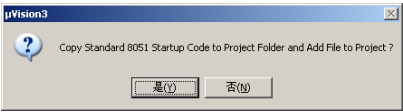


图 6-3 提示信息

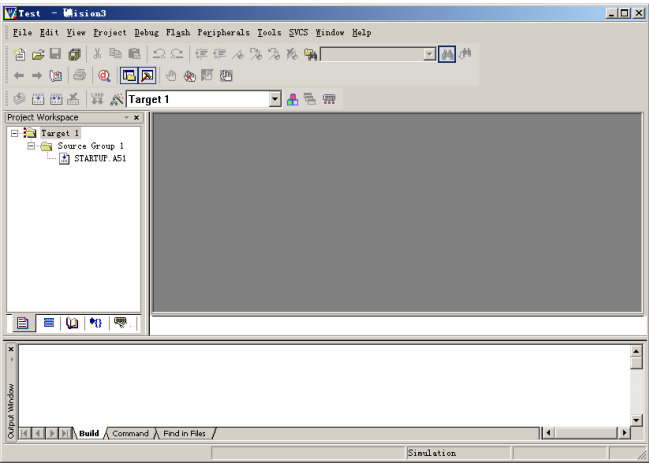


图 6-4 项目建立完毕

6.2.2 创建源文件

当一个单片机项目建立完毕后,现在开始进行核心的源文件的设计。具体的操作步骤如下:

(1) 选择“File”→“New”命令,此时工作区中弹出一个新的文本编辑窗口,如图 6-5 所示。

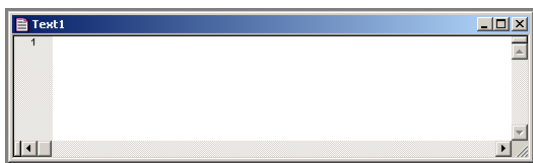


图 6-5 新建的文本编辑窗口

(2) 在其中输入下列程序代码。

```
#include <reg51.h>                                //头文件

sbit sel=P0^0;                                     //位定义

void main()                                        //主函数
{
    P1=0x01;                                       //P1 端口预置数
    while(1)                                      //循环执行程序
    {
        if (sel==1)                               //判断端口数值,如果为高电平
        {
            P1=P1>>1;                             //P1 端口数据右移
            if (P1==0x00)                           //P1 端口为零时,重新置数
                P1=0x80;                             //以便于循环操作
        }
        else
        {
            P1=P1<<1;                             //P1 端口数据左移
            if (P1==0x00)                           //P1 端口为零时,重新置数
                P1=0x01;                             //以便于循环操作
        }
    }
}
```

这段程序给出了一个完整的 C51 语言的程序代码。其功能是判断单片机 P0.0 端口的电平。如果为高电平,就让 P1 口循环输出高电平;如果为低电平,就让 P1 口反方向循环输出高电平。

从这段程序代码可以看出,C51 语言和 C 语言的程序结构基本一致,包括头文件、主函数等。程序中用到的循环语句、判断语句的结构都完全一样。因为 C51 语言运行于单片机系统,P0 和 P1 为单片机的端口资源,C51 语言可以直接对端口操作,这是有别于 C 语言的地方。

另外,这段代码不用做任何修改,便可以运行于所有的 51 系列单片机。这也充分体现了 C51 语言相比于汇编语言的优越性。

(3) 代码输入完毕后,可以单击“保存”按钮,保存为 test.c 文件。

(4) 在项目管理窗口中,右键单击“Source Group 1”,选择“Add Files to Group ‘Source Group 1’”命令,在弹出的对话框中选择刚才保存的 C 语言源文件,并加入项目中即可。

6.2.3 编译项目

项目及源文件建立完毕后,便可以编译项目了。选择“Project”→“Build target”命令,即可编译,如果程序无误,则在输出窗口中显示编译结果,如图 6-6 所示。



图 6-6 编译输出结果

如果需要生成单片机上可执行的文件，可以选择“Project”→“Options for Target ‘Target 1’”命令，此时弹出“Options for Target ‘Target 1’”对话框，如图 6-7 所示。在“Output”选项卡中，选择复选框“Create Hex File”，并单击“确定”按钮保存设置。

此时，重新编译一次，便可以生成可以下载到单片机中的执行文件 Test.hex；然后可以利用下载工具将其下载到单片机中执行。

6.2.4 仿真调试

当一个单片机项目建立并编译通过后，在下载至单片机硬件中之前，为了确保程序的执行完全符合要求，一般需要进行仿真调试。μ Vision3 的集成开发环境提供了完善的项目仿真和调试工具，下面就介绍如何使用 μ Vision3 来进行单片机 C 语言程序的仿真调试。项目仿真调试之前需要确保该项目完全通过编译。仿真调试的步骤如下：

(1) 选择“Debug”→“Start/Stop Debug Session”命令，此时，μ Vision3 进入仿真调试模式。界面的菜单栏和工具栏都相应地发生了变化。

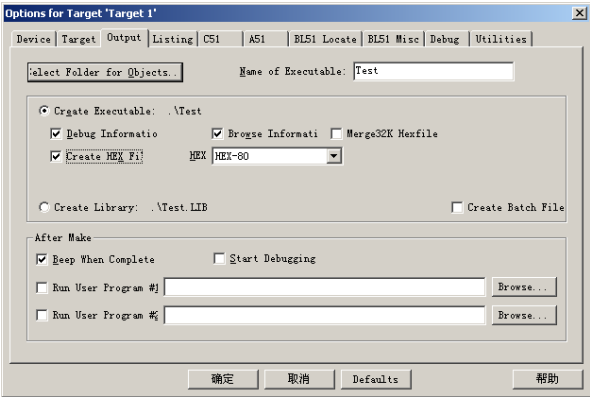


图 6-7 “Options for Target ‘Target 1’”对话框

(2) 选择“Peripherals”→“I/O-Ports”→“Port 0”命令，打开并行端口 0 仿真窗口，如图 6-8 所示。

(3) 同样，选择“Peripherals”→“I/O-Ports”→“Port 1”命令，打开并行端口 1 仿真窗口，如图 6-9 所示。

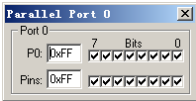


图 6-8 并行端口 0 仿真窗口

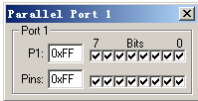


图 6-9 并行端口 1 仿真窗口

(4) 此时，使用单步执行的方式运行程序。每按一次 F11 键，程序执行一条指令。用户可以看到程序运行时，单片机的并行端口 1 循环输出高电平。

(5) 如果置 P0.0 端口为低电平，则并行端口 1 反方向循环输出高电平。

以上介绍的是使用 Keil μVision3 集成开发环境进行单片机 C 语言程序设计、编译及仿真调试的基本步骤。在本书后面的章节会逐步介绍单片机 C 语言及其程序设计，读者可以在学习中采用同样的方法对程序进行仿真调试，从而加快学习速度。

6.3 C51 的标识符与关键字

同标准的 C 语言一样，单片机的 C51 程序语言也包含特定的标识符和关键字。因此，在进行 C51 程序设计时必须符合其使用的规则。

6.3.1 标识符

标识符是用来标识源程序中某个对象名称的符号。其中的对象可以是常量、变量、语句标号、数据类型、自定义函数名及数组名等。C51 标识符的定义不是随意的，需要符合以下定义规则。

- C51 的标识符可以由字母、数字（0~9）和下画线“_”组成。
- C51 的标识符区分大小写，例如“num5”和“NUM5”代表两个不同的标识符。
- C51 的标识符第一个字符必须是小写字母（a~z）、大写字母（A~Z）或下画线“_”。例如“count1”、“C_1”等，都是正确的。而“5num”则是错误的标识符，在编译时系统会出现错误提示。另外，有些编译系统专用的标识符是以下画线开头，为了程序的兼容性和可移植性，所以建议一般不要以下画线开头来命名标识符。
- C51 的标识符定义不能使用 C51 的关键字，也不能和用户已使用的函数名或 C51 库函数同名。例如“int”是不正确的标识符，“int”是关键字，所以它不能作为标识符。
- C51 的标识符最多可支持 32 个字符，不过，为了使用和理解方便，尽量不要使用过长的标识符。

另外，在命名 C51 标识符时，应当尽量简单，并且能够清楚地表达含义，这样有助于阅读和理解源程序。

6.3.2 关键字

关键字是被 C51 编译器已定义保留的专用特殊标识符。关键字是 C51 语言的一部分，如 if、for、do、case 等。这些关键字有固定的名称和含义，用户在 C51 源程序中自定义的标识符不允许与关键字相同，否则程序将无法编译运行。单片机 C51 程序语言采用了 ANSI C 标准定义的 32 个关键字，这些关键字如表 6-1 所示。

表 6-1 ANSI C 标准的关键字

关 键 字	类 型	用途说明
auto	存储种类说明	常用于说明局部变量，默认值为此
break	程序语句	无条件退出程序最内层循环
case	程序语句	switch 语句中的选择项
char	数据类型说明	单字节整型数据或字符型数据
const	存储类型说明	定义不可更改的常量值
continue	程序语句	中断本次循环，并转向下一次循环
default	程序语句	switch 语句中的默认选择项
do	程序语句	用于构成 do...while 循环结构
double	数据类型说明	定义双精度浮点型数据
else	程序语句	构成 if...else 选择程序结构
enum	数据类型说明	枚举
extern	存储种类说明	在其他程序模块中说明了的全局变量

续表

关 键 字	类 型	用途说明
Float	数据类型说明	定义单精度浮点型数据
for	程序语句	构成 for 循环语句
goto	程序语句	构成 goto 转移结构
if	程序语句	构成 if...else 选择结构
int	数据类型说明	基本整型数据

long	数据类型说明	长整型数据
register	存储种类说明	CPU 内部寄存的变量
return	程序语句	用于返回函数的返回值
short	数据类型说明	短整型数据
signed	数据类型说明	有符号数，二进制表示的最高位为符号位
sizeof	运算符	计算表达式或数据类型的占用字节数
static	存储种类说明	定义静态变量
struct	数据类型说明	定义结构类型数据
switch	程序语句	构成 switch 选择结构
typedef	数据类型说明	重新定义数据类型
union	数据类型说明	联合类型数据
unsigned	数据类型说明	定义无符号数据
void	数据类型说明	定义无类型数据
volatile	数据类型说明	该变量在程序执行中可被隐含地改变
while	程序语句	用于构成 do...while 或 while 循环结构

另外，不同的编译系统会根据 C51 语言及单片机硬件的特点来扩展相关的关键字。在 Keil μ Vision3 编译器中采用的扩展关键字，如表 6-2 所示。

表 6-2 C51 扩展的关键字

关 键 字	类 型	用途说明
bit	位标量声明	声明一个位变量或位类型的函数
sbit	位标量声明	声明一个可位寻址的变量
sfr	特殊功能寄存器声明	声明一个 8 位的特殊功能寄存器
sfr16	特殊功能寄存器声明	声明一个 16 位的特殊功能寄存器
data	存储器类型说明	直接寻址的单片机内部数据存储器
bdata	存储器类型说明	可位寻址的单片机内部数据存储器
idata	存储器类型说明	间接寻址的单片机内部数据存储器
pdata	存储器类型说明	分页寻址的单片机内部数据存储器
xdata	存储器类型说明	单片机外部数据存储器
code	存储器类型说明	单片机程序存储器
interrupt	中断函数说明	定义一个中断函数
reentrant	再入函数说明	定义一个再入函数
using	寄存器组定义	定义单片机的工作寄存器

6.4 C51 的数据类型

C51 的数据类型是 C51 语言中变量及常量的类型。每个变量在使用之前必须定义其数据类型。C51 除了继承了标准 C 语言中基本的数据类型 int、char、short、long、float 和 double 等外，又有自己的特点。例如在 C51 语言中 int 和 short、float 和 double 具有相同的取值范围和含义。


在 C51 中有以下几种基本数据类型：整型（int）、浮点型（float）、字符型（char）、无值型（void）。此外，C51 语言还提供了几种聚合类型（aggregate types），包括数组、指针、结构、联合（共用体）、枚举和位域。关于几种聚合类型将在下一章中进行详细介绍。本章主要介绍几种基本类型。

C51 语言中基本数据类型的字长和取值范围，如表 6-3 所示。

表 6-3 基本数据类型的字长和取值范围

类 型	字长（bit）	取值范围
char（字符型）	8	0~255
int（整型）	16	-32 768~32 767

float（单浮点型）	32	约精确到 6 位数
void（无值型）	0	无值

说明：表中各个基本数据类型的字长和取值范围是假定 CPU 的字长为 16bit。


在基本数据类型中，除 void 类型外，其前面均可以有各种修饰符。修饰符用来改变基本类型的意义，以便更准确地适应各种情况的需求。修饰符如下：

- signed（有符号）
- unsigned（无符号）
- long（长型符）
- short（短型符）

其中，修饰符 signed、short、long 和 unsigned 可以修饰字符型和整型两种基本类型，而修饰符 long 还可用于 double 型。C51 中所有根据 ANSI 标准而组合的数据类型的字长和取值范围，如表 6-4 所示。

表 6-4 ANSI 标准中的数据类型

类 型	字 长（bit）	取值范围
char（字符型）	8	ASCII 字符
unsigned char（无符号字符型）	8	0~255
signed char（有符号字符型）	8	-128~127
Int（整型）	16	-32 768~32 767
unsigned int（无符号整型）	16	0~65 535
Signed int（有符号整型）	16	同 int
char（字符型）	8	ASCII 字符
Short（短整型）	8	-128~127
unsigned short int（无符号短整型）	8	0~255
signed short int（有符号短整型）	8	同 short int
long int（长整型）	32	-2 147 483 648~2 147 483 649
signed long int（有符号长整型）	32	-2 147 483 648~2 147 483 649
unsigned long int（无符号长整型）	32	0~4 294 967 296
float（单浮点型）	32	约精确到 6 位数

说明：表中各个数据类型组合的字长和取值范围是假定 CPU 的字长为 16bit。

因为整数类型的默认定义是有符号数，所以 signed 修饰符经常可以省略。某些编译环境允许将 unsigned 用于浮点型，如 unsigned double。但这一用法并不通用，降低了程序的可移植性，故建议程序设计时一般不要采用。

为了使用方便，C51 编译程序允许使用如下的整型简写形式。

- short int 简写为 short
- long int 简写为 long
- unsigned short int 简写为 unsigned short
- unsigned int 简写为 unsigned
- unsigned long int 简写为 unsigned long

6.4.1 常量与变量

常量就是在程序执行过程中不改变值的数据量，变量就是在程序运行过程中可以改变值的数据量。常量和变量是计算机程序处理的数据对象。

1. 常量

在程序中，常量是不接受程序修改的固定值，C51 中的常量可为任意数据类型。示例如下：

```
char          'c','\n','5'           //字符型常量
int           10、123、210、-134      //整型常量
short int     12、-12、90             //短整型常量
unsigned int  10000、987、4           //无符号整型常量
long int      31500、-34              //长整型常量
float        145.23、4.34e-3         //浮点型常量
```

C51 语言还支持字符串常量，这是另一种预定义数据类型的常量。所有字符串常量括在双撇号之间，例如"This is a string test"。这里需要注意的是，不要把字符常量和字符串常量相混淆，单个字符常量是由单撇号括起来的，如'a'。

2. 变量

在程序中，变量是可以被程序修改的数据量。一个完整的变量应该有类型说明符及变量标识符。C51 编译器在单片机内存中单独划分一定的存储单元，用来存放该变量的值。

所有的 C51 变量必须在使用之前定义。定义变量的一般形式是“类型说明符 变量标识符，变量标识符...”。示例如下：

```
Int           i,j,k;                  //整型变量
short int     si;                     //短整型变量
unsigned int   unt;                   //无符号整型变量
float         fset, pro_fit,lost;     //浮点型变量
```

6.4.2 整型数据

整型数据是整数类型的数据。整型数据是最常用的数据类型，下面就介绍整型数据的说明、整型变量的定义及整型常量的表示。

1. 整型数据说明

整型数据包括整型变量和整型常量两种，其前面可以加上不同的修饰符，整型数据有以下几种类型。

- unsigned short int: 无符号短整型数说明，简写为 unsigned int。无符号短整型字长为 2 字节共 16 位二进制数，数据的范围是 0~65 535。
- signed short int: 有符号短整型数说明，简写为 short 或 int。有符号短整型字长为 2 字节共 16 位二进制数，数据的范围是-32 768~32 767。
- unsigned long int: 无符号长整型数说明，简写为 unsigned long。无符号长整型字长为 4 字节共 32 位二进制数，数据的范围是 0~4 294 967 295。
- signed long int: 有符号长整型数说明，简写为 long。有符号长整型字长为 4 字节共 32 位二进制数，数据的范围是-2 147 483 648~2 147 483 647。

在有符号型数据的二进制表示中，字节最高位表示数据的符号，“0”表示正数，“1”表示负数。

2. 整型变量定义

定义整型变量的一般格式是“类型说明符 变量标识符，变量标识符...”。示例如下：

```
int           a, b;                   //定义 a、b 为有符号短整型变量
unsigned long  c, d;                   //定义 c、d 为无符号长整型变量
```

其中，类型说明符与变量标识符之间至少有一个空格。在最后一个变量标识符后必须以“;”结尾。另外，在 C51 中定义变量时，允许同时定义多个相同类型的变量，各变量间用逗号间隔。变量定义必须放在变量使用之前，一般位于函数的开头。示例程序如下：

```
#include <stdio.h>                    //头文件
```

```

void main( )                                //主函数
{
    int      a,b,c,d;                      //定义 a , b , c , d 为整型变量
    unsigned  x;                            //定义 x 为无符号整型变量
    a=10; b= -22; x=20;                    //赋初值
    c=a+x; d=b+x;                          //变量运算
    printf("a+x=%d, b+x=%d\n",c,d);        //打印输出结果
}

```

该程序可以在 Keil μ Vision3 集成开发环境中执行，运行结果如下：

```
a+x=30, b+x=-2
```

在本例中，int 型数据与 unsigned int 型数据进行相加减运算，可以看到不同类型的整型数据可以进行算术运算。

3. 整型常量表示

整型常量及整型常数可以表示十进制、八进制、十六进制的整数值。按不同的进制区分，整型常数有如下三种表示方法。

- 十进制数：以非 0 开始的数来表示。例如 220, -560, 45 900。
- 八进制数：以 0 开始的数来表示，起始 0 是必须的引导符。例如 06, 0 106, 05 788。
- 十六进制数：以 0X 或 0x 开始的数来表示，其中引导符 0 是必须有的，X 即字母既可用大写也可用小写。例如 0X0D, 0XFF, 0x4e。

在整型常数后添加一个“L”或“l”字母，则表示该数为长整型数。例如 17L, 0753L, 0Xfa7l 等。

整型常量在不加特别说明时总是正值。如果需要的是负值，则负号“-”必须放置于常数表达式的最前面。例如 -0x48, -07。

6.4.3 浮点型数据

浮点型数据是用于表示包含小数点的数据类型。下面分别介绍在 C51 语言中浮点型数据的说明、浮点型数据的定义及浮点型常量的表示。

1. 浮点型数据说明

C51 语言中支持如下三种类型的浮点数，即 float 类型、double 类型和 long double 类型。只不过，在 C51 语言中不具体区分这三种类型，都被当做 float 类型对待。因此，这三种浮点类型有相同的精度和取值范围。浮点类型的变量字长为 4 个字节共 32 位二进制数，取值范围为 $3.4 \times 10^{-38} \sim 3.4 \times 10^{+38}$ 。另外，在 C51 语言中，浮点型数据均为有符号浮点数，而没有无符号浮点数。

2. 浮点型变量定义

浮点型变量的一般定义格式是“类型说明符 变量标识符，变量标识符...”。同整型数据一样，浮点型变量也可以同时定义多个。

示例如下：

```

float      a, b;                            //定义 a、b 为单精度浮点型变量
double     c, d;                            //定义 c、d 为双精度浮点型变量

```

示例程序如下：

```

#include <stdio.h>                          //头文件
void main( )                                //主函数
{
    float    a;                              //定义 a 为单精度浮点型变量

```



```
double b; //定义 b 为双精度浮点型变量
a=1134.5678; //赋值
b=5890.1;
printf("a=%f\nb=%f\n",a,b); //打印输出结果
}
```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
a=1134.568000
b=5890.100000
```

在本例中，a 为单精度浮点型变量，有效位数是 7 位，因此最后四舍五入为 1134.568。b 为双精度浮点型变量，在 C51 中按 float 型处理，有效位数也是 7 位。

3. 浮点型常量表示

浮点型常量通常称为实型常量，其值可以表示为十进制。浮点型常量的值由整数部分、尾数部分和指数部分组成。一般情况下，在不加说明的情况下，浮点型常量为正值。如果表示负值，需要在常量前使用负号。例如 28.57、-57.63、-7.2e-16、8.265。

在 C51 语言中，所有浮点常数都被默认为 float 型。对于绝对值小于 1 的浮点数，其小数点前面的零可以省略。例如 0.27 可写为.27、-0.031E-4 可写为-.031E-4。

在编译环境中采用默认格式输出浮点数时，最多只保留小数点后六位，不够的后面补零。在浮点型常量中不得出现任何空白符号。字母 E 或 e 之前必须有数字，表示形式为“数字 e \pm 数字”，且浮点型常量中 E 或 e 后面指数必须为整数，例如“e2.1”、“e3”等都是不合法的指数形式。

6.4.4 字符型数据

字符型变量是用来存放字符的变量类型。字符型变量只能存放一个字符。在信息的传递中，字符、文字等是经常用到的。下面介绍 C51 中的字符型数据的定义和使用等。

1. 字符型变量定义

在 C51 语言中，可以定义有符号字符变量和无符号字符变量两种类型的字符型变量。字符型变量的定义格式示例如下：

```
char a; //定义 a 为有符号字符变量
unsigned char b; //定义 b 为无符号字符变量
```

在 C51 语言中，字符型数据在操作时将按整型数据处理，计算机中的字符是以 ASCII 码方式表示的，其长度为 1 个字节。因此有符号字符型变量的范围为-128~127，无符号字符型变量的值范围是 0~255。如果某个变量定义成 char，则表明该变量是有符号的，即它将转换成有符号的整型数据。示例如下：

```
#include <stdio.h> //头文件

void main() //主函数
{
    char c1,c2,c3,c4; //定义字符变量
    c1=97; //赋值
    c2=98; //赋值
    c3='a';
    c4='b';
    printf("c1=%c\nc2=%c\nc3=%c\nc4=%c",c1,c2,c3,c4); //输出结果
}
```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
c1=a
c2=b
```

```
c3=a
c4=b
```

程序中, c1、c2、c3、c4 被指定为字符变量。其中, c3 和 c4 分别赋值两个字符, 因此输出结果为字符。在程序中, 将十进制整数 97 和 98 分别赋给 c1 和 c2, 它的作用也相当于以下两个赋值语句。

```
c1 = 'a';
c2 = 'b' ;
```

字符 a 和 b 的 ASCII 码为 97 和 98, 因此程序中使用 “%c” 来表示表示输出两个字符。

在 C51 语言中, 如果需要存放一个字符串, 则需要定义一个字符型数组, 将字符串存到该数组中来表示。示例如下:

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    char str[3];                                    //定义字符型数组
    str [0]='A';                                    //赋值
    str [1]='B';                                    //赋值
    printf("str=%s",str);                          //打印输出结果
}
```

在 C51 编译时, 系统将留出三个字符的连续空间, 即 str [0]~str [2]共三个变量, 其中最后一个用来存放字符串终止符 NULL, 即 “\0”, 只有前两个可以自由赋值。

这段程序可在 Keil μ Vision3 集成开发环境中运行, 执行结果如下:

```
str=AB
```

这里需要注意的是, 终止符是编译程序时自动加上的, 程序中只对前两个赋值, 最后一个不用赋值。如果对 str[2]也进行赋值, 将导致程序的错误输出。

2. 字符常量表示

字符常量是指用一对单引号括起来的一个字符。其中单引号只起定界作用, 并不表示字符本身, 例如 'F', '7', '!' 等。在 C51 语言中, 字符常量单引号中的字符不能是单引号 (') 和反斜杠 (\)。如果需要使用这两个字符, 则需要采用转义字符来表示, 这将在后面介绍。

在 C51 语言中, 字符是按其所对应的 ASCII 码值来存储的, 一个字符占一个字节。因此也可用该字符的 ASCII 码值表示, 例如十进制数 85 表示大写字母 'U', 十六进制数 0x5d 表示符号 ']', 八进制数 0101 表示大写字母 'A' 等。

另外, 一些不能用符号表示的控制符, 也可以用 ASCII 码值来表示。例如十进制数 13 表示回车符、十六进制数 0x0A 表示换行符、八进制数 033 表示 Esc 等。这些控制符是无法在程序中显式地表示的。

由于 C51 语言中字符常量是按整数 (short 型) 存储的, 所以字符常量可以像整数一样在程序中参与相关的运算。示例如下:

```
'a'-32 ;                                //执行结果 97-32 = 65
'A'+32;                                //执行结果 65+32 = 97
'8'-8;                                //执行结果 56-8 = 48
```

这里值得强调的是字符 '9' 和数字 9 的区别, 前者是字符常量, 后者是整型常量, 它们的含义和在计算机中的存储方式都截然不同。

下面举一个例子来演示字符常量以及字符常量的运算, 程序示例如下:

```
#include <stdio.h>                                //头文件
```

```
void main()                                //主函数
{
    char c1,c2;                            //定义字符变量
    c1='a';                                //赋值
    c2='b';
    c1=c1-32;                              //转换大小写
    c2=c2-32;
    printf("c1=%c\nc2=%c",c1,c2); //输出结果
}
```

这段程序可在 Keil μVision3 集成开发环境中运行，执行结果如下：

```
c1=A
c2=B
```

本例是将两个小写字母转换为大写字母。因为'a'的 ASCII 码为 97，而'A'为 65，'b'的 ASCII 码为 98，'B'则为 66。从 ASCII 代码表中可以看到每一个小写字母比对应大写字母的 ASCII 码大 32。即'a'='A'+32。

3. 转义字符

转义字符用于表示 ASCII 码字符集中不可打印的控制字符和特定功能的字符。转义字符是 C51 语言中表示字符的一种特殊形式。例如可以用于表示字符常量的单引号 (')，用于表示字符串常量的双引号 (") 和反斜杠 (\) 等。转义字符用反斜杠 “\” 后面跟一个字符或一个八进制或十六进制数表示，如表 6-5 所示。

表 6-5 转义字符

转义字符	含 义	ASCII 码值（十进制）
\a	响铃（BEL）	007
\b	退格（BS）	008
\f	换页（FF）	012
\n	换行（LF）	010
\r	回车（CR）	013

续表

转义字符	含 义	ASCII 码值（十进制）
\t	水平制表（HT）	009
\v	垂直制表（VT）	011
\\	反斜杠	092
\?	问号字符	063
\'	单引号字符	039
\"	双引号字符	034
\0	空字符（NULL）	000
\ddd	任意字符	三位八进制
\xhh	任意字符	二位十六进制

如果需要在字符常量中使用单引号和反斜杠及字符常量中使用双引号和反斜杠时，都必须使用转义字符表示，即在这些字符前加上反斜杠。

在 C51 语言中，可以使用转义字符\ddd 或\xhh 方便灵活地表示任意字符。其中，\ddd 为斜杠后面跟三位八进制数，该三位八进制数的值即为对应的八进制 ASCII 码值；\x 后面跟两位十六进制数，该两位十六进制数为对应字符的十六进制 ASCII 码值。

使用转义字符时需要注意以下几点。

- 在 C51 程序中，使用不可打印字符时，通常用转义字符来表示。
- 转义字符中只能使用小写字母，每个转义字符只能看做一个字符。
- \v 垂直制表和\f 换页符在屏幕中的显示没有任何影响，但是会影响打印机的打印操作。

6.4.5 指针型数据

指针型数据是指向变量所存放地址的数据类型。指针型数据是一种特殊的数据类型，根据所指的变量类型不同，可以分为整型指针、浮点型指针、字符型指针、结构型指针和联合指针。在C51语言中，指针型变量的定义格式如下：

```
int      *i;           //定义整型指针变量
float    *f;           //定义浮点型指针变量
char     *c;           //定义字符型指针变量
struct   *stu;         //定义结构型指针变量
union    *uni;         //定义联合指针变量
```

下面以字符型指针变量为例介绍指针型数据的使用，程序示例如下：

```
#include <stdio.h>      //头文件

void main()             //主函数
{
    char ch;            //定义字符型变量
    char *p;            //定义字符型指针
    ch='b';             //变量赋值
    p=&ch;              //将变量c的地址赋给p
    printf("p=%c", *p); //输出地址p中所存的数据内容
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
*p=b
```

该程序中，首先给字符型变量 `ch` 赋值 'a'，然后将 `ch` 的地址赋给字符型指针 `p`。这样 `p` 所指向的地址存放的数据是 'b'。

6.4.6 无值型数据

无值型数据是一个特殊的类型，其字节长度为 0。在 C51 语言中，无值型主要有以下两个用途。

- 定义一个同一类型指针，该指针可根据需要动态分配内存。
- 在定义和使用函数时，明确地表示这个函数不返回任何值。

无值型数据的声明示例如下：

```
void      *buf;         //buf 被定义为无值型指针
```

6.5 C51 的变量作用域

变量作用域是变量在程序中起作用的范围。C51 语言中，使用变量前需要首先声明该变量。由于 C51 程序中可以包含多个函数和程序文件，因此使用变量时，需要注意变量有效作用范围的问题，即变量的作用域。

6.5.1 基本规则

在 C51 语言中规定，在块结构中进行定义的变量，其有效使用范围只在该块结构内。其中，块结构是任何以花括号括起来的一段程序，通常称为复合语句。在 C51 程序中，一些函数和语句是以一对花括号 “{” 和 “}” 来构成的，这也是块结构的一种，示例如下：

```
#include <stdio.h>      //头文件
```

```
void main() //主函数
{
    int i; //定义整型变量
    i=23; //变量赋值
    if(i%2==0) //判断是否是偶数，如果是执行下面的语句
    {
        int j; //定义整型变量
        j=2; //变量赋值
    }
    printf("i=%d",i); //输出变量 i，正确
    printf("j=%d",j); //输出变量 j，错误
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行。其中，整型变量 i 定义在主函数的大括号内，其使用范围为整个主函数，因此，后面的打印输出语句正确。而变量 j 定义在复合语句 if 内，因此其只在该 if 块结构中使用，出了该块结构便无效了。因此，最后的打印输出语句是错误的，在编译程序时，会提示该变量未定义。

由于块结构内部的变量和块结构外部的变量作用域不同，因此，若块结构内定义的变量与块结构外定义的变量具有相同的变量名，它们之间不冲突，示例如下：

```
#include <stdio.h> //头文件
void main() //主函数
{
    int i; //定义整型变量
    i=23; //变量赋值
    if(i%2==0) //判断是否是偶数，如果是执行下面的语句
    {
        int i; //定义整型变量
        i=2; //变量赋值
        printf("in if block i=%d\n",i); //输出变量 i
    }
    printf("out if block i=%d\n",i); //输出变量 i
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行，程序输出结果如下：

```
in if block i=2
out if block i=23
```

该程序在主函数中定义了整型变量 i，然后在 if 块结构中也定义了 i，因为其作用域不同，因此，在 if 语句内部，其内部定义的变量起作用，外部定义的变量被屏蔽。在 if 语句外部，是主函数定义变量的有效范围。

变量作用域即变量的作用范围，其可以是一个函数，一个程序文件，甚至整个工程里的所有文件都可用。一般而言，按照存储类型，变量分为 4 种类型，即自动变量、外部变量、静态变量和寄存器变量。下面分别介绍其作用域的范围。


6.5.2 自动变量

自动变量是以关键字 auto 标识的变量类型。其声明格式为“[auto] 类型说明符 变量标识符，变量标识符...”。自动型变量可以在如下两种情况下使用。

- 在函数的内部进行定义，其作用域的范围为该函数的内部，在该函数之外变量无效。
- 在程序块中声明，其作用域的范围为该程序块。

在编译 C51 程序时，自动型变量根据变量类型动态分配存储空间。动态分配的过程是，在

程序执行到该函数时，为其自动分配存储空间，当该函数执行完毕后，立即结束该变量的存储空间，即该自动型变量消失。这便是为什么在函数内部定义的变量，不能在函数外引用的原因。

 **说明：**在 C51 程序中，函数或程序块内部定义的变量，一般都默认为自动型变量。

因此，在声明自动型变量时，关键字 auto 可以省略。

使用自动变量的程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int i=0;                                       //定义整型变量
    int j=1;                                       //定义整型变量
    {                                             //程序块 1 开始
        int i=1;                                   //定义整型变量
        int j=2;                                   //定义整型变量
        {                                       //程序块 2 开始
            int i=2;                               //定义整型变量
            int j=3;                               //定义整型变量
            printf("i=%d,j=%d\n",i,j);             //输出
        }                                         //程序块 2 结束
        printf("i=%d,j=%d\n",i,j);               //输出
    }                                             //程序块 1 结束
    printf("i=%d,j=%d",i,j);                     //输出
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
i=2,j=3
i=1,j=2
i=0,j=1
```

在该程序中，主函数声明了整型变量 i 和 j，然后分别有两个块结构的复合语句，其中分别定义并初始化变量 i 和 j。按照 C51 语言规定，各个变量默认为自动型变量，因此其作用域仅限于函数内部和块结构内部，不会互相影响。

6.5.3 外部变量

外部变量一般定义在所有函数的外部，即整个程序文件的最前面，也称为全局变量。外部变量的作用域是整个程序文件，可以被该程序文件中的任何函数使用。

在编译 C51 程序时，外部变量根据变量类型被静态地分配适当的存储空间。在整个程序运行过程中，该变量一旦分配空间，便不会消失。这也是外部变量对整个程序文件有效的原因。

因为外部变量是永久性的，因此使用外部变量，可以作为不同函数间参数的传递和共享。示例如下：

```
#include <stdio.h>                                //头文件

double PI=3.14;                                   //圆周率 PI
int r;                                             //整型变量代表半径

double circle()                                    //计算面积函数
```

```
{
    return r*r*PI;           //返回圆的面积
}
void main()                 //主函数
{
    r=2;                    //半径赋值
    printf("the area is %f\n",circle()); //输出圆的面积
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
the area is 12.560000
```

该程序在源文件的头部定义了两个外部变量 **PI** 和 **r**，分别表示圆周率和半径。整型变量 **r** 在主函数中初始化，然后在 **circle** 函数中调用，用来计算面积。

如果一个 C51 工程项目由多个程序文件组成时，外部变量允许在一个程序文件中定义，而在另一个程序文件中使用。这样，在使用时，需要在文件的头部用关键字 **extern** 来显式地进行声明。这样 C51 在编译的时候，才能正确知道该外部变量的数据类型及值。

上面的程序可以采用下面这种方法来实现，将程序分为 **circle.c** 和 **main.c** 两个文件。其中，**circle.c** 程序文件代码如下：

```
#include <stdio.h>           //头文件

double PI=3.14;
int r;

double circle()
{
    return r*r*PI;
}
```

主程序 **main.c** 文件代码如下：

```
#include <stdio.h>           //头文件

extern PI;
extern r;

double circle();

void main()                 //主函数
{
    r=2;
    printf("the area is %f\n",circle());
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
the area is 12.560000
```

该程序实现的功能和前面程序是一样的，只不过分为两个文件来实现。在 **circle.c** 文件中定义了外部变量 **PI** 和 **r**，在主文件 **main.c** 中使用它们之前，先用 **extern** 加以声明，这样便可以使用 **circle.c** 文件中定义的外部变量了。

6.5.4 静态变量

静态变量以关键字 **static** 声明，声明格式为“**static** 类型说明符 变量标识符，变量标识符...”。在 C51 语言中，根据声明的位置不同，静态变量可以分为如下两种。

- 内部静态变量，在函数内部定义；
- 外部静态变量，在函数外部定义。

在编译 C51 程序时,根据数据类型静态地给静态变量分配合适的存储空间,并在程序运行过程中始终占有该存储空间。

另外,C51 语言允许将函数定义为静态类型的。这样,只有同一文件中的其他函数才能调用这个静态型函数,同一工程项目中的其他文件不能调用访问,这样,可以既有利于程序的模块化设计,又可以防止和其他文件中的函数有重名的情况。



说明:从变量作用域来看,内部静态变量和自动变量类似,作用域只是定义该变量的函数内部。从占用内存的角度,内部静态变量和全局变量类似,其始终占有内存空间。

6.5.5 寄存器变量

寄存器变量以关键字 `register` 声明,声明格式为“`register` 类型说明符 变量标识符, 变量标识符...”。示例如下:

```
register char    ch;
register double  db;
```

寄存器变量常用于某一变量名频繁使用的情况,这样做可以提高系统的运算速度。因为普通的变量一般存储在内存中,而寄存器变量被存储在 CPU 的寄存器中。

在 C51 语言中,只允许同时定义两个寄存器变量,如果多于两个,程序在编译时会自动地将两个之外的寄存器变量当做非寄存器变量来处理。

6.6 分隔符与 const 修饰符

分隔符和修饰符是计算机语言中最重要的组成部分,这里介绍 C51 语言中的分隔符和 `const` 修饰符。

6.6.1 C51 分隔符

分隔符在 C51 程序语言中起辅助标识和分界的作用。一般分为两种,简单分隔符和复合分隔符。简单分隔符是除字母、数字、下画线以外的单个 ASCII 字符,如+、-、*、/、!、|、<、>、=、.、:、;、'、#等。复合分隔符是特定字符的某种组合,例如<=、!=、>=、{、}、[、]、(、)等。

分隔符常用于数组、语句等情况,下面举出常用的分隔符示例。

- 用“;”进行隔离语句,例如 `int r;`
- 用“[]”对数组说明,例如 `char ch[]="Hello Word!";`
- 用“()”进行隔离,例如 `s=area(r);`
- 用“,”进行隔离,例如 `char ch1, ch2, ch3;`
- 用“{}”进行隔离复合语句,例如 `for {i=0;i<100;i++};`
- 用“*”做指针,例如 `char *ponter;`
- 用“#”做预处理伪指令,例如 `#define PI 3.1415926;`
- 用“^”标识特殊寄存器的位,例如 `sbit P00=P0^0。`

6.6.2 const 修饰符

`const` 修饰符主要用来定义常量或变量。其定义格式为“`const` <类型说明符> <常量名> =<常量值>;”。示例如下:

```
const double PI=3.1415926;
```

在 C51 语言中,用关键字 `const` 修饰的是一类特殊的常量,一般称为符号常量或 `const` 变量。从编译的角度看,程序在编译时,将视 `const` 变量为一个常量,系统不为其分配内存。在程序中遇到该 `const` 变量时,将用定义时的初值来代替。

这里需要注意的是,使用 `const` 修饰的变量不能在程序中修改其值,这样一方面可以防止程序

51 单片机开发与应用技术详解

运行时该值被意外修改，另一方面也可以对于程序中经常使用的值做统一修改，便于调试程序。
下面举一个例子。

```
#include <stdio.h>                                //头文件

const double PI=3.14;                             //定义圆周率为 const 变量

void main()                                        //主函数
{
    int r;                                         //定义半径
    double area,length;                          //定义圆面积和周长
    r=2;                                          //半径赋值
    area=r*r*PI;                                 //计算面积
    length=2*PI*r;                               //计算周长
    printf("the area=%lf,length=%lf\n",area,length); //输出结果
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
the area=12.560000,length=12.560000
```

该程序中，将圆周率定义为 double 型的 const 变量。主程序执行时，在遇到该变量时自动以 double 型的值 3.14 来代替。

另外，在 C51 语言中，宏定义和 const 变量有些类似。宏定义在后面的章节中会具体介绍，这里仅指出其与 const 变量的区别。

宏定义在程序预处理时只对上下文进行简单的文本代替，并不做具体的语法检查。示例如下：

```
#define count 20;
```

程序在编译时，将遇到的字符串 count 全部替换为字符串 20。这在使用时会有一些问题，如这个变量是整型数据还是浮点型数据呢，很容易引起混乱。同样是定义这个常量，使用 const 变量，可以如下定义。

```
const int          count=20;                      //定义 count 为整型
const double count=20;                            //定义 count 为浮点型
```

这样，就很容易区分其数据类型了。因此在这种情况下，使用 const 变量要比宏定义简洁清楚。

6.7 运算符

运算符是一个表示特定的算术或逻辑操作的符号，也称为操作符。例如“+”号，表示了一个相加运算；“&&”号表示了一个逻辑与的运算。在 C51 语言中，由运算符把需要进行运算的各个量（常量或变量）连接起来组成一个表达式。

C51 语言中的运算符很丰富，主要有三大类运算符，即算术运算符、关系与逻辑运算符、位操作运算符。另外，还有一些用于完成复杂功能的特殊运算符。

表 6-6 C51 语言的算术运算符

运 算 符	含 义
-	减法、取负
+	加法
*	乘法
/	除法
%	取模运算
--	自减（减 1）
++	自增（增 1）

6.7.1 算术运算符

算术运算符是用来进行算术运算的操作符。C51 语言中允许的算术运算符，如表 6-6 所示。C51 语言中的运算符“+”、“-”、“*”和“/”的用法与大多数计算机语言相同，几乎可用于所有 C51 语言内定义的数据类型。

1. 普通算术运算

普通算术运算主要包括加 (+)、减 (-)、乘 (*)、除 (/), 以及取模 (%) 的运算。示例如下:

```
6+1=7;
9-5=4;
2*3=6;
6/2=3;
7/4=1;
21%2=1;
```

其中加减乘除运算和其他语言的运算没什么特别之处, 这里需要强调下面几点。

- 除法运算 (/) 是取除法结果的整数部分。
- 取模运算 (%) 是取除法结果的余数部分。另外该运算符不能应用于浮点型数据的操作。
- 减法运算符 (-) 可以用做取负操作, 如 -sz 是取变量 sz 的负操作。

这里举例说明这些运算符在程序中的使用, 程序示例如下:

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int x,y,z;                                    //定义整型变量
    x=20;y=11;                                    //变量赋初值
    z=x+y;                                        //加法运算
    printf("x+y=%d\n",z);                        //输出结果
    z=x-y;                                        //减法运算
    printf("x-y=%d\n",z);                        //输出结果
    z=x*y;                                        //乘法运算
    printf("x*y=%d\n",z);                        //输出结果
    z=x/y;                                        //除法运算
    printf("x/y=%d\n",z);                        //输出结果
    z=x%y;                                        //取模运算
    printf("x%%y=%d\n",z);                      //输出结果
    z=-x;                                        //取负运算
    printf("-x =%d\n",z);                       //输出结果
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行, 执行结果如下:

```
x+y=31
x-y=9
x*y=220
x/y=1
x%y=9
-x=-20
```

2. 自增和自减运算

自增运算符 “++” 表示操作数加 1, 自减运算符 “--” 表示操作数减 1。C51 语言中这两个很有用的运算符是继承了 C 语言的特点。换句话说, $x=x+1$ 等同于 $x++$; $x=x-1$ 等同于 $x--$ 。

在大多数的编译环境中, 采用自增和自减操作符所生成的程序代码, 比等价的赋值语句生成的代码执行得要快, 因此一般推荐采用自增和自减运算符。

自增和自减运算符可用在操作数之前, 也可放在其后, 例如 $x=x+1$, 可写成 $++x$, 也可以写成 $x++$, 但在表达式中这两种用法是有区别的。自增或自减运算符在操作数之前时, C51 语

言在引用操作数之前就先执行加 1 或减 1 操作；运算符在操作数之后时，C51 语言就先引用操作数的值，而后再进行加 1 或减 1 操作。示例如下：

```
x=++t; //t 先增 1 后赋值给 x
x=t++; //t 先赋值给 x 然后再增 1
```

下面举一个采用自增和自减运算符的例子，程序示例如下：

```
#include <stdio.h> //头文件

void main() //主函数
{
    int x,y,z1,z2; //定义整型变量
    x=20;y=11; //赋初值
    z1=(x++)+(x++);
    z2=(++y)+(++y);
    printf("%d,%d,%d,%d",x,y,z1,z2); //输出结果
}
```

这段程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

22,13,41,25

在该程序中，x 自增两次，最后值为 22。y 自增两次，最后值为 13。在 C51 语言中，按自左向右的运算顺序，z1 是 x 和 x 增 1 相加，所以是 41；z2 是 y 增 1 和 y 继续增 1 相加，所以是 25。

3. 算术运算符的优先级

算术运算符的优先级由高到低依次为自增自减（++、--）和取负（-）、乘法除法（*、/）和取模（%）、加和减（+、-）。

这里需要强调的是，在 C51 程序编译时对同级运算符一般按从左到右的顺序进行计算，由于括号的优先级最高，所以括号会改变计算顺序。

6.7.2 逻辑运算符和关系运算符

逻辑运算符中的“逻辑”描述了操作数的逻辑关系，而关系运算符中的“关系”描述了一个操作数与另一个操作数之间的比较关系。关系运算符和逻辑运算符通常在一起使用，所以这里将它们放在一起进行讲解。

1. 逻辑运算符

逻辑运算符是进行逻辑运算的操作符。逻辑运算符主要有三种，如表 6-7 所示。逻辑运算符的操作对象可以是整型数据、浮点型数据，以及字符型数据。逻辑运算符的逻辑真值表，如表 6-8 所示。如果逻辑运算符的操作结果是真，则为 1，如果为假则为 0。

表 6-7 C51 语言的逻辑运算符

运 算 符	含 义
!	逻辑非运算
	逻辑或运算
&&	逻辑与运算

表 6-8 C51 语言的逻辑真值表

A	B	A&&B	A B	!A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

下面举一个例子，说明逻辑运算符在程序设计中的应用。

```
#include <stdio.h> //头文件

void main() //主函数
{
```

```
int a,b,c,d; //定义整型变量，存储逻辑运算结果
a=!0; //逻辑非运算
b=10&&20; //逻辑与运算
c=5&&0; //逻辑与运算
d=2.1||0; //逻辑或运算
printf("a=%d,b=%d,c=%d,d=%d\n",a,b,c,d); //输出结果
}
```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
a=1,b=1,c=0,d=1
```

这里需要指出的是，在 C51 语言中规定非零的操作数都为真，为零的操作数为假。

2. 关系运算符

关系运算符主要用于比较操作数的大小关系，在 C51 语言中和一般的 C 语言类似。常用的关系运算符有以下几种，如表 6-9 所示。

下面举一个例子，来说明关系运算符在程序设计中的应用。

表 6-9 C51 语言的逻辑运算符

```
#include <stdio.h> //头文件

void main() //主函数
{
    int a,b,c,d; //定义整数变量，存储结果
    a>0>-7.2; //比较运算
    b=20==30; //比较运算
    c=5!=0; //比较运算
    d=0<=0; //比较运算
    printf("a=%d,b=%d,c=%d,d=%d\n",a,b,c,d); //输出结果
}
```

运 算 符	含 义
>	大于
>=	大于或等于
<	小于
<=	小于或等于
==	等于
!=	不等于

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
a=1,b=0,c=1,d=1
```

3. 返回值

逻辑运算符和关系运算符的返回值都是 True（真）和 Flase（假）。在 C51 语言中规定，非 0 的值为 True，0 值为 Flase。使用关系或逻辑运算符的表达式时，若表达式为 True（真）则返回值为 1；若表达式为 Flase（假）则返回值为 0。

4. 关系运算符和逻辑运算符的优先级

关系运算符和逻辑运算符的优先级符合如下几点。

- 关系运算符和逻辑运算符的相对优先级最高的是！，其次是>、<、>=和<=，然后是==和!=，后面是&&，最后是||。
- 关系和逻辑运算符的优先级比算术运算符低，即像表达式 10>1+12 和表达式 10>(1+12) 运算的结果是一样的。
- 在一个表达式中允许运算的组合。
- 同算术运算符一样，在关系或逻辑表达式中也可以使用括号来修改原运算顺序。
- 所有关系和逻辑表达式产生的结果不是 0 就是 1。示例如下：

```
11>5&&!(10<8)||3<=7
```

这一表达式的结果为 True，即返回值为 1。

！注意：括号的处理方法与其他的计算机语言相同，主要用于强制将某个运算符或某组

运算的优先级升高。

6.7.3 位运算符

位运算是对于字节或字中的二进制位（bit）进行测试、置位、移位或逻辑处理的运算符。这里字节或字是针对 C 标准中的 char 和 int 数据类型而言的，位操作不能用于 float、double、long double、void 或其他复杂类型。

支持全部的位运算符（Bitwise Operators）是 C51 语言与其他高级语言最大的不同，即具有汇编语言所具有的运算能力。因此 C51 既具有高级语言的特点，也具有低级语言的功能。

C51 语言中的位运算符，如表 6-10 所示。位运算中的 AND、OR 和 NOT（1 的补码）的真值表与逻辑运算等价，唯一不同的是位操作是逐位进行运算的。下面分别介绍各个位运算符的用法。

1. 按位与运算符

按位与运算符是将两个操作数按二进制展开，对应位进行逻辑与的运算符。按位与运算符“&”是二目运算符，要求有两个操作数。如果对应的二进制位均为 1，则逻辑与的结果为 1；只要有其中一个为 0，则逻辑与的结果为 0。

例如将 a=100 和 b=6 进行按位与运算。100 的二进制表示为 01100100，6 的二进制表示为 00000110。结果 a&b=4（00000100）。

注意：区分位运算和关系逻辑运算，关系逻辑运算结果不是 0 就是 1；而位运算的结果通过相应处理，结果可为任意值。例如 a=100 和 b=6，a&b=4，而 a&&b=1。

2. 按位或运算符

按位或运算符是将两个操作数按二进制展开，对应位进行逻辑或的运算符。按位或运算符“|”也是二目运算符。如果对应的二进制位均为 0，则逻辑或的结果为 0；只要有其中一个为 1，则逻辑或的结果为 1。同按位与运算符一样，两个操作数的或运算可以是任意值。

例如将 a=100 和 b=6 进行按位或运算。100 的二进制表示为 01100100，6 的二进制表示为 00000110。结果 a|b=102（01100110）。

3. 按位非运算符

按位非运算符是将操作数按二进制展开，每一位取反，即将 0 变为 1，将 1 变为 0。按位非运算符“~”是一目运算符，用来对一个数求反。

例如将 a=100 进行按位取反运算。100 的二进制表示为 01100100，结果 ~a=155（10011011）。

4. 按位异或运算符

按位异或运算符是将两个操作数按二进制展开，对应位进行逻辑异或的运算符。按位异或运算符“^”也是二目运算符。如果对应的二进制位均为 0 或均为 1，则逻辑或的结果为 0；只要对应位不同，则逻辑或的结果为 1。异或的真值表，如表 6-11 所示。

表 6-10 C51 语言的位运算符

运 算 符	作 用
&	逻辑与（AND）
	逻辑或（OR）
^	逻辑异或（XOR）
~	按位取补（NOT）
>> <<	右移 左移

例如将 a=100 和 b=6 进行按位或运算。100 的二进制表示为 01100100，6 的二进制表示为 00000110。结果 a^b=98（01100010）。

表 6-11 异或的真值表

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

5. 移位运算符

移位运算符有两种, 分别将操作数的各位按要求向右或向左移动。移位运算符的使用格式如下:

- 右移语句形式是: 变量名 (或操作数) >> 右移位数
- 左移语句形式是: 变量名 (或操作数) << 左移位数

需要注意的是, C51 语言中的移位不是循环移位, 当某位从一端移出时, 另一端移入 0。从一端移出的位并不送回到另一端去, 移去的位永远丢失了, 同时在另一端补 0。

注意右移运算, 对无符号位数左端补 0, 称为“逻辑右移”; 如果为负数, 即符号位为 1, 则左端补 1, 这种补 1 保持负号的方法称为“算术右移”。例如 x=7, 将其进行移位操作, 结果如表 6-12 所示。

表 6-12 用移位操作进行乘和除

字符 x	每个语句执行后的 x	x 的值
x=7	00000111	7
x<<1	00001110	14
x<<3	01110000	112
x<<2	11000000	192
x>>1	01100000	96
x>>2	00011000	24

从表 6-12 中可以看出, 每左移一位, 结果乘 2, 注意 x<<2 后, 原 x 的信息已经丢失了, 因为一位“1”已经从一端移出。每右移一位相当于被 2 除。

移位操作可实现对外部设备 (如 D/A 转换器) 的输入和状态信息的译码, 移位操作还可用于整数的快速乘除等运算。

6. 程序实例

前面具体介绍了各个位运算符, 下面举一个例子, 来演示一下位运算符在程序中的具体应用。程序代码示例如下:

```
#include <stdio.h>                                     //头文件

void main()                                             //主函数
{
    int a,b,z;                                         //定义变量
    a=57;                                              //变量赋值
    b=221;
    z=a&b;                                             //按位与运算
    printf("a&b=%d\n",z);                             //输出结果
    z=a|b;                                             //按位或运算
    printf("a|b=%d\n",z);                             //输出结果
    z=a^b;                                             //按位异或运算
    printf("a^b=%d\n",z);                             //输出结果
    z=~a;                                              //按位取反运算
    printf("~a=%d\n",z);                             //输出结果
    z=a<<2;                                           //左移两位
    printf("a<<2=%d\n",z);                             //输出结果
    z=b>>3;                                           //右移三位
    printf("b>>3=%d\n",z);                             //输出结果
}
```

该程序可在 Keil μ Vision3 集成开发环境中运行, 执行结果如下:

```
a&b=25
a|b=253
a^b=228
~a=-58
a<<2=228
b>>3=27
```

读者可以参考前面介绍的内容，分析一下程序运行的结果和预期的是否一致。

6.7.4 特殊运算符

除了前面几节介绍的几种运算符外，在 C51 语言中还有一些特殊运算符，用于一些复杂的运算，可以起到简化程序的作用。

1. “,” 运算符

“,” 运算符是把几个表达式串在一起，按照顺序从左向右计算的运算符。“,” 运算符左侧的表达式不返回值，只有最右边的表达式的值作为整个表达式的返回值。示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int a,b,z;                                    //定义变量
    a=57;                                          //赋值
    b=221;
    z=(a++,++b+a);                                //利用“,”运算符为 z 赋值
    printf("z=%d\n",z);                          //输出结果
}
```

该程序可在 Keil μVision3 集成开发环境中运行，执行结果如下：

```
z=280
```

本例中首先执行 a++，a 自增 1，然后执行 ++b+a，并将结果赋给变量 z。因为逗号操作符的优先级比赋值操作符优先级低，所以这里必须使用括号。

2. “?” 运算符

“?” 运算符是三目操作符，其一般形式如下：

```
EXP1?EXP2:EXP3;
```

其中，EXP1、EXP2 和 EXP3 是表达式，注意冒号的用法和位置。

“?” 运算符作用是在计算表达式 EXP1 的值后，如果其值为 True，则计算表达式 EXP2 的值，并将其结果作为整个表达式的结果；如果表达式 EXP1 的值为 False，则计算表达式 EXP3 的值，并将其作为整个表达式的结果。这个运算符在某些场合代替 if - then - else 语句，可以起到简化程序的作用。

“?” 运算符在程序中的使用示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int x,y;                                    //定义变量
    x=11;                                          //赋值
    y=x>9?100:200;                                //使用“?”运算符为 y 赋值
    printf("y=%d\n",y);                          //输出结果
}
```

本例中，首先判断 x>9 为真，所以将第一个数 100 赋给 y。如果 x 是比 9 小的值，则 y 的

值将为 200。若用 if-else 语句改写，有下面的等价程序，从中可以看出使用“?”运算符，可以大大简化程序的编写。

```
#include <stdio.h>                                //头文件
void main()                                       //主函数
{
    int x,y;                                     //定义变量
    x=10;                                         //赋值
    if (x>9)                                     //使用 if 语句为 y 赋值
    {
        y=100;
    }
    else
    {
        y=200;
    }
    printf("y=%d\n",y);                          //输出结果
}
```

这两段程序都可以在 Keil μVision3 集成开发环境中运行，读者可以自己进行仿真。

3. 地址操作运算符

地址操作运算符主要有“*”和“&”两种。

“&”运算符是一个单目操作符，其返回操作数的地址。“*”运算符和“&”运算符相对应，也是一目操作符，其返回位于某个地址内存储的变量值。示例如下：

```
#include <stdio.h>                                //头文件

main()                                           //主函数
{
    char a,b;                                   //定义字符型变量
    char *p;                                    //定义指针型变量
    a='a';                                       //为字符型变量 a 赋值
    p=&a;                                        //将变量 a 的地址赋给 p
    b=*p;                                       //地址 p 所指的单元值赋给 b
    printf("b=%c\n",b);                        //输出 b
}
```

该程序可在 Keil μVision3 集成开发环境中运行，执行结果如下：

```
b=a
```

本例中，字符型变量 a 赋值为'a'，然后将其的地址赋给指针型变量 p，最后将地址 p 中的数值赋给变量 b。这样变量 b 便有了和变量 a 一样的内容。

4. 联合操作

联合操作主要用来简化一些特殊的赋值语句，这类赋值语句的一般形式如下：

```
<变量 1>=<变量 1><操作符><表达式>
```


利用联合操作可以简化如下：

```
<变量 1><操作符>=<表达式>
```

使用联合操作符的语句示例如下：

```
a=a+b      可简化为 a+=b
a=a*b      可简化为 a*=b
```


a=a&b 可简化为 a&=b
a=a|b 可简化为 a|=b
a=a/(x+y-z) 可简化为 a/=x+y-z

说明：这种联合操作适用于所有双目操作符，是从 C 语言中继承来的。

5. “sizeof” 运算符

“sizeof” 运算符是单目操作符，其返回变量所占的字节或类型长度字节。在 C51 语言中，“sizeof” 运算符其实更像一个函数，类似于 C 语言中 length 函数。使用 “sizeof” 运算符的程序示例如下：

```
#include <stdio.h>                                     //头文件

void main()                                           //主函数
{
    char a[]="hello world!";                         //定义字符串数组
    int i,j;                                           //定义整型变量
    i=sizeof(a);                                     //获得字符串数组的长度
    j=sizeof(double);                                //获得 double 类型数据的长度
    printf("i=%d\nj=%d\n",i,j);                      //输出结果
}
```

该程序可在 Keil μVision3 集成开发环境中运行，执行结果如下：

i=13
j=4

6. 类型转换运算符

类型转换运算符用于强制使某一表达式变为特定类型，为一目运算符，并且同其他一目操作符的优先级相同。其一般形式如下：

(类型) 表达式

其中（类型）中的类型必须是 C51 中的一个数据类型。

类型转换运算符的使用示例如下：

(float)x/2

其中为确保表达式 x/2 的结果具有类型 float，所以使用类型转换运算符强制转换为浮点型数据。类型转换运算符在程序中的使用示例如下：

```
#include <stdio.h>                                     //头文件

void main()                                           //主函数
{
    int i;                                           //定义整型循环变量
    for(i=0;i<5;i++)                                //for 循环语句
    {
        printf("%d/2=%f\n",i,(float)i/2);          //循环输出 i/2 的数值
    }
}
```

该程序可在 Keil μVision3 集成开发环境中运行，执行结果如下：

0/2=0.000000
1/2=0.500000
2/2=1.000000
3/2=1.500000
4/2=2.000000

6.7.5 运算符优先级和结合性

在 C51 语言中，当一个表达式中有多个运算符参与运算时，要按照运算符的优先级别进行运算。在一个复杂的表达式中，常常有许多运算符和变量，除了要判断优先级还要考虑结合性（或者关联性）这个因素。示例如下：

```
-5+7;
```

这里的表达式需要用结合性来判断，因为运算符“-”和“+”相对于运算的操作数来说是“左”结合的，所以实际参与计算的是“-5”和“+7”，运算的结果为 2。

这里需要强调的是“左结合性”和“右结合性”的含义。

- 左结合性：变量（或常量）与左边的运算符结合。
- 右结合性：变量（或常量）与右边的运算符结合。

运算符优先级和结合性，如表 6-13 所示，表中优先级从上往下逐渐降低，同一行优先级从左往右逐渐降低。

表 6-13 运算符的优先级和结合性

优先级	运算符（高—————→低）	结 合 性	运算符类型
（ 高 ↓ 低 ）	()[]->.	从左至右	双目运算符
	! ~ ++-- （类型） sizeof +- * &	从右至左	单目运算符
	*/%	从左至右	双目运算符
	+ -	从左至右	双目运算符
	<< >>	从左至右	双目运算符
	< <= > >=	从左至右	双目运算符
	== !=	从左至右	双目运算符
	&	从左至右	双目运算符
	^	从左至右	双目运算符
		从左至右	双目运算符
	&&	从左至右	双目运算符
		从左至右	双目运算符
	?:	从右至左	三目运算符
	= += -= *= /= %= &= ^= = <<= >>=	从左至右	双目运算符

从表 6-13 中可以看出一个规律，凡是单目运算符都是“右结合”的，凡是双目运算符都是“左结合”的。其中有“?”和“:”运算符是三目运算符是右结合的，记住了这个规律，运算符的结合性也就掌握了。

实际程序设计中，如果代码行中的运算符比较多，由于将表熟记是比较困难的。所以，为了防止产生歧义并提高可读性，应当多用括号确定表达式的操作顺序，避免使用默认的优先级。示例如下：

```
(num<<8) | a //用括号把需要先运算的部分括起来
(a|b) && (a&c) //用括号把需要先运算的部分括起来
```

下面举一个完整的例子，程序示例如下：

```
#include<stdio.h> //头文件

void main() //主函数
{
    int a=2; //定义整型变量
    a%=4-1; //联合操作
    printf("a=%d\n",a); //输出结果
    a+=a*=a-=a*=3; //复杂的运算
```

51 单片机开发与应用技术详解

```
printf("a=%d\n",a);           //输出结果
}
```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
a=2
a=0
```

本例中由于 $\% =$ 运算符的优先级别低于 $-$ 运算， $a\%=4-1$ 即是 $a\%=3$ ，等价于 $a=a\%3$ 即 $a=2\%3=2$ 。表达式 $a+=a*=a-=a*=3$ ，开始时 $a=2$ ，表达式赋值是从左至右进行的，表达 $a*=3$ 使得 $a=6$ ，此表达式的值也为 6，于是表达式 $a-=a*=3$ 相当于 $a=6-6=0$ ， a 的值为 0，后面的计算就不用做了，所以 a 的值最终为 0。

6.8 表达式

表达式是由运算符把需要进行运算的各个量连接起来而构成的一个整体。表达式主要由操作数和运算符组成。操作数一般包括常量和变量，有时甚至可以包括函数和表达式等。同运算符一样，表达式也是 C51 语言中的基本组成部分。

6.8.1 算术表达式

算术表达式是指用算术运算符和括号将操作数连接起来，并且符合 C51 语法规则的式子。例如 $a+(b-c)*2-'b'$ ，这是一个正确的算术表达式。算术表达式比较简单，主要应该注意算术运算符的计算顺序。下面举一个例子，来演示算术表达式的应用，程序示例如下：

```
#include <stdio.h>           //头文件

void main()                   //主函数
{
    int i,j,x,y;              //声明变量
    i=23;
    j=12;
    x=i+j;                    //算术运算
    y=i-j;                    //算术运算
    printf("i+j=%d\ni-j=%d\n",x,y); //输出结果
}
```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```
i+j=45
i-j=11
```

6.8.2 赋值表达式

赋值表达式是指由赋值运算符将一个变量和一个表达式连接起来的式子，其一般形式如下：

```
<变量><赋值运算符><表达式>
```

例如 “ $x=15$ ” 就是一个简单的赋值表达式，表示将 15 赋值给变量 x 。赋值表达式的求解过程是将赋值运算符右边的表达式的值赋给左边的变量。赋值表达式在程序中的应用示例如下：

```
#include <stdio.h>           //头文件

void main()                   //主函数
{
    int i,j;                  //声明变量
    char a,b;
```

```

a='c';           //变量赋值
b='d';
i=15+'a';        //赋值
j=b-'D'+a;
printf("i=%d\nj=%d\n",i,j); //输出结果
}

```

该程序可在 Keil μ Vision3 集成开发环境中运行, 执行结果如下:

```

i=112
j=131

```

由于在 C51 语言中, 字符型变量和整型变量具有相同的取值范围。所以本例中, 字符型变量可以和整型数据一起参与运算。

赋值表达式中需要特别注意的是数据类型转换的问题。数据类型转换是指不同类型的变量混用时的类型之间的转化。在赋值语句中, 数据类型转换的规则是等号右边的值转换为等号左边的值所属的类型。示例如下:

```

#include <stdio.h>           //头文件

void main()                 //主函数
{
    int a,b,k;              //声明变量
    float f;
    a=2;                    //变量赋值
    b=3;
    k=a*b;                  //i 与 j 的乘积为整型数据, 赋值给 k 时仍为整型
    f=a*b;                  //因 f 为浮点型数据, 赋值时自动转化为浮点型
    printf("k=%d\nf=%f\n",k,f); //输出结果
}

```

该程序可在 Keil μ Vision3 集成开发环境中运行, 执行结果如下:

```

k=6
f=6.000000

```

由于赋值运算时的特殊性, 在做除法运算时应特别注意数据类型之间的转换。示例如下:

```

#include <stdio.h>           //头文件

void main()                 //主函数
{
    int i;                  //声明变量
    float f;
    i=15;                   //赋值
    f=i/2;                  //i/2 为整型, 只保留了整数部分, 即 i/2=7
    printf("f=%f\n",f);     //输出结果
    f=i/2.0;               //此时, i/2.0=7.5
    printf("f=%f\n",f);     //输出结果
}

```

该程序可在 Keil μ Vision3 集成开发环境中运行, 执行结果如下:

```

f=7.000000
f=7.500000

```

在本程序中, $i/2$ 按照除法的运算规则, 除不尽时取整, 因此 $f=7.000000$; 而对于 $i/2.0$, 操作数有一个为浮点型, 因此, 计算的结果为 7.500000 , 保留了正确结果。在程序设计时一定要注意这种情况。

6.8.3 逗号表达式

逗号表达式是用逗号运算符将两个表达式连接起来的式子。其一般形式如下：

表达式 1, 表达式 2, 表达式 3, ... 表达式 n

逗号表达式的应用示例如下：

```
#include <stdio.h>                                //头文件

void main()                                         //主函数
{
    int a,b;                                       //声明变量
    b=(a=3*10,a*8);                               //逗号表达式
    printf("a=%d\nb=%d\n",a,b);                 //输出结果
}
```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

a=30
b=240

本例中，先求解 $a=3*10$ ，则 a 的值为 30，再求解 $a*8$ ，则整个表达式的值为 240，然后将 240 赋值给 b。一个逗号表达式还可以与另一个表达式组成新的逗号表达式。示例如下：

(a=3*10 , a*8) , (a+10)

本例中，由左边表达式得 $a=30$ ，再进行 $a+10$ 的运算，得整个表达式的值为 40，与左边表达式的值 240 无关。

在使用逗号表达式时，需要注意以下几点。

- 程序中使用逗号表达式，有时并不一定为了求整个逗号表达式的值，而可能需要分别求出逗号表达式内各表达式的值。示例如下：

(a=5*6 , a*8) , (a+10);

整个逗号表达式没有赋值给任何变量，其主要是计算各个中间表达式的值。

- 逗号表达式中的表达式 n 也可以是逗号表达式，这样便构成嵌套结构，计算的顺序仍然是自左向右，整个表达式的值为最后一个表达式的值。
- 在程序中，并不是所有出现逗号的地方都是逗号表达式，例如在变量声明、函数参数表中的逗号只是用做变量之间的分隔符。示例如下：

int a,b,c;

6.8.4 关系和逻辑表达式

关系和逻辑表达式是采用关系运算符及逻辑运算符来构成的式子。关系和逻辑表达式常用于程序控制语句中控制流程运算。关系表达式和逻辑表达式通常是结合在一起使用。

1. 关系表达式

关系表达式是指用关系运算符将两个表达式连接起来的式子。关系运算又称为“比较运算”。示例如下：

$x \leq y, x \neq z, (x > 5) \geq 0$

关系表达式的计算结果是逻辑值，即“真”（True）和“假”（False）。当结果为真时，表达式的值为 1，反之为 0。示例如下：

```
#include <stdio.h>                                //头文件

void main()                                         //主函数
{
```

```

int a,b,c,d;           //声明变量
a=4;                   //赋值
b=5;
c=(a>b);               //计算关系表达式
d=(a!=b);
printf("c=%d\nd=%d\n",c,d); //输出结果
}

```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```

c=0
d=1

```

2. 逻辑表达式

逻辑表达式是指用逻辑运算符将两个表达式连接起来的式子。逻辑表达式中的运算对象可以是任何类型的数据，如字符型、整型或指针型等。

逻辑表达式的值是逻辑值，即“真”和“假”。C51 语言中在给出逻辑运算结果时，以数值 1 代表“真”，以数值 0 代表“假”。示例如下：

```

#include <stdio.h>           //头文件

void main()                 //主函数
{
    int a,b,c,d;            //声明变量
    a=4;                    //变量赋值
    b=5;
    c=a||b;                 //计算逻辑表达式
    d=!a;
    printf("c=%d\nd=%d\n",c,d); //输出结果
}

```

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

```

c=1
d=0

```

注意：在逻辑表达式的运算中，系统并不执行所有的运算符。只是在必须执行下一个逻辑运算符才能求解表达式时，才执行该运算符，否则，将不执行该运算符。

3. 在控制流程语句中的应用

关系和逻辑表达式的一个主要用途是用于控制程序的流程，通常需要配合 if、while 和 for 等语句来完成。下面给出一个简单的例子，程序代码示例如下：

```

#include <stdio.h>           //头文件

void main()                 //主函数
{
    int a,b,c;              //声明变量
    a=3;                    //赋值
    b=0;
    if(a&&b)                 //判断 a&&b=0，为假
        c=a||b;
    else
        c=!a;                //执行此语句 c=!a=0
    printf("c=%d\n",c);      //输出结果
}

```

51 单片机开发与应用技术详解

该程序可在 Keil μ Vision3 集成开发环境中运行，执行结果如下：

c=0

6.9 小结

单片机 C 语言是现在单片机系统设计所广泛采用的程序语言。本章首先介绍了单片机 C 语言（C51 语言）的特点，然后分别介绍了 C51 语言的标识符、关键字、数据类型、运算符和表达式等，并对每一个知识点都提供了完整详细的实例分析。本章是单片机 C51 语言的基础，熟练掌握本章知识，对以后章节的学习会有很大的帮助。

第 7 章 数组

数组是把具有相同数据类型的若干变量按有序形式组织起来的集合。其中，数组中的单个变量称为数组元素。数组属于构造数据类型。一个数组可以分解为多个数组元素，这些数组元素可以是基本数据类型或构造类型。按照数组元素所属的基本数据类型的不同，数组又可分为数值数组、字符数组、指针数组、结构数组等各种类别。在 C51 语言中，数组可以是一维的，也可以是多维的。

7.1 数组类型说明

在 C51 语言中，使用数组前必须先进行类型说明。数组说明的一般形式如下：

类型说明符 数组名 [常量表达式] , ;

其中，类型说明符可以是任一种基本数据类型或构造数据类型；数组名是用户定义的数组标识符，相当于变量名；方括号中的常量表达式用于表示数据元素的个数，也称为数组的长度。典型的数组声明，示例如下：

```
int count[10];           //声明整型数组 count，共 10 个元素
float b[10],c[20];       //声明浮点型数组 b，共 10 个元素，浮点型数组 c，共 20 个元素
char ch[20];             //声明字符数组 ch，共 20 个元素
```

对于数组类型声明应注意以下几点。

- 数组名的命名规则应符合上一章中介绍的标识符的命名规定。
- 对于一个数组，其内部所有元素的数据类型都是相同的。数组的类型实际上是指数组元素的数据类型。
- 数组名不能与其他变量名相同。例如，下面的数组声明是错误的。

```
void main()
{
    int ch;                //声明整型变量
    float ch[10];          //声明整型数组，命名冲突
    ...
}
```

- 方括号中常量表达式表示数组元素的总个数，例如 a[5]表示数组 a 有 5 个元素。在 C51 语言中，数组的下标从 0 开始计算的，因此 5 个元素分别为 a[0]、a[1]、a[2]、a[3]、a[4]。
- C51 语言不支持动态分配数组大小，因此不能在方括号中用变量来表示元素的个数，但是可以使用符号常数或常量表达式。示例如下：

```
#define FD 5              //宏定义
void main()
{
    int a[3+2],b[7+FD];    //声明整型数组
```


...

这段程序中，首先宏定义 **FD** 为 **5**，然后可以在数组中使用 **FD** 来表示 **5** 了。但是如下的声明方式是错误的，因为其中使用了整型变量来表示数组大小。

```
void main()
{
    int n=5;                //声明并初始化整型变量
    int a[n];               //声明整型数组，数组大小不能为变量
    ...
}
```

➤ 在 C51 语言中，可以在同一个类型说明中声明多个数组和多个变量。示例如下：

```
int a,b,x,y,k1[10],k2[20];
```

其中 **a**、**b**、**x** 和 **y** 为整型变量，**k1** 和 **k2** 是整型数组，数组大小分别为 **10** 和 **20**。

7.2 数组元素的表示

数组元素是组成数组的基本单元。在 C51 语言中，数组元素也是一种变量，其标识方法为数组名后跟一个下标。下标表示了元素在数组中的顺序号。数组元素的一般形式如下：

数组名[下标]

这里的下标只能为整型常量或整型表达式。例如，**a[6]**、**a[i+]**、**a[i++]**都是合法的数组元素。数组元素通常也称为下标变量。在 C51 语言中必须先定义数组，才能使用下标变量。另外，程序中只能逐个地使用下标变量，而不能一次引用整个数组。

例如，输出有 **20** 个元素的数组必须使用循环语句，逐个输出各下标变量对应的元素值。

```
for(i=0; i<20; i++)
{
    printf("a[%d]=%d",i,a[i]);    //循环输出
}
```

而不能用一个语句输出整个数组，下面的写法是错误的。

```
printf("a=%d",a);
```

另外，和声明数组时不同，在引用单个数组元素时，可以使用变量来表示下标标号，但该变量必须有初始值。这里举一个完整的例子，演示数组元素在程序中的使用。

```
#include <stdio.h>                //头文件

void main()                       //主函数
{
    int i,a[10];                  //定义整型变量 i 和整型数组 a
    for(i=0;i<10;i++)             //循环赋值
        a[i]=2*i+1;
    for(i=0;i<10;i++)             //循环输出结果
        printf("%d ",a[i]);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
1 3 5 7 9 11 13 15 17 19
```

在本程序中，首先定义整型变量 **i** 和整型数组 **a**，其中 **a** 有 **10** 个数组元素。接着，利用 **for** 循环语句为数组 **a** 中的每个元素分别赋值。最后，将数组中的元素逐个输出。

7.3 数组元素的初始化赋值

数组元素的初始化赋值是指在数组声明的时候给数组元素赋予初值。数组初始化赋值是在编译阶段进行的，这样将减少程序运行时间，提高效率。数组元素初始化赋值的一般形式如下：

类型说明符 数组名 [常量表达式]={值, 值.....值};

在{ }中的各数据值即为各元素的初值，各值之间用逗号间隔。示例如下：

```
int a[10]={0,1,2,3,4,5,6,7,8,9};
```

该语句相当于赋值语句 $a[0]=0, a[1]=1, \dots, a[9]=9$ 。

在 C51 语言中，对数组的初始赋值需要注意以下几点。

➤ 在 C51 语言中，只能给元素逐个赋值，不能给数组整体赋值。

例如，给 10 个元素全部赋 1 值，只能写为如下形式。

```
int c[10]={1,1,1,1,1,1,1,1,1,1};
```

而不能写为如下形式。

```
int c[10]=1;
```

➤ 可以只给部分元素赋初值。当{ }中值的个数少于元素个数时，只给数组的前面部分元素赋值。示例如下：

```
int a[10]={0,1,2,3,4};
```

该例表示只给 $a[0] \sim a[4]$ 5 个元素赋值，而后 5 个元素自动赋 0 值。注意，此时应该在方括号中给出数组的长度，否则将按数组长度为 5 处理。

➤ 如果是给数组的全部元素赋初始值，则在数组声明中，可以不给出数组元素的个数。

示例如下：

```
char a[5]={'a','b','c','d','\0'};
```

可等价地写为如下形式。

```
char a[]={'a','b','c','d','\0'};
```

➤ 如不给可初始化的数组赋初值，C51 会自动处理。此时，外部型和静态型变量的初始值为 0，自动型和寄存器型变量的初始值为随机数。

➤ 动态赋值可以在程序执行过程中，对数组作动态赋值。这时可用循环语句配合 scanf 函数逐个对数组元素赋值。程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int i,a[4];                                    //定义整型变量 i 和整型数组 a
    printf("请输入 4 个整数\n");                  //打印说明字符
    for(i=0;i<4;i++)                              //循环输入数组 a 中的元素
    {
        scanf("%d",&a[i]);
    }
    for(i=0;i<4;i++)                              //循环输出数组 a 中的元素
    {
        printf("a[%d]=%d ",i,a[i]);
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在本程序中，首先使用 scanf 语句和 for 语句，循环输入整型数组 a 中的 4 个元素。最后，使用 printf 语句，逐个输出数组元素。

7.4 一维数组

数组是一个由若干同类型变量组成的集合，引用这些变量时可用同一名字。数组均由连续的存储单元组成，最低地址对应于数组的第一个元素，最高地址对应于最后一个元素，数组可以是一维的，也可以是多维的。一维数组是指只有一个下标标号的数组。

7.4.1 一维数组声明

在 C51 语言中，一维数组的一般说明形式如下：

类型说明符 数组名 [常量表达式]；

如果在声明数组时就给数组进行初始化赋值，可以采用如下的形式。

类型说明符 数组名 [常量表达式]={值, 值, ..., 值}；

一维数组声明示例如下：

```
int a[4]={1,2,3,4}
```

本例定义了整型数组 a[4]，包含 a[0]、a[1]、a[2]、a[3]共 4 个元素。数组中元素与位置的对应关系，如图 7-1 所示。

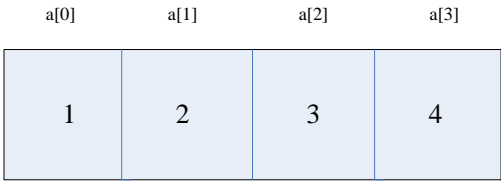


图 7-1 数组的构成

为了便于编译程序而为数组预先分配内存空间，数组大小必须显示地说明。在 C51 语言中，一维数组的总字节数可按下式计算。

sizeof(类型)*数组长度＝总字节数

一维数组在本质上是由同类型数据构成的顺序表结构，例如，下面的一维数组 a。

```
int a[7];
```

假定起始地址为 0x1000，数组 a 在单片机内存中的存放情形，如表 7-1 所示。

表 7-1 起始地址为 0x1000 的 7 元素字符数组

数组元素	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
地 址	0x 1000	0x 1001	0x 1002	0x 1003	0x 1004	0x 1005	0x 1006

7.4.2 向函数传递一维数组

在 C51 语言中，将一维数组传递给函数时，并不是将整个数组作为实参来传递，而是用指针来代替。因此，只需把数组名作为参数直接调用函数即可，无须指定任何下标。这样，数组的第一个元素的地址将传递给该函数。例如，以下代码将整型数组 ch 的第一个元素的地址传递给函数 function1()。

```
void main( )
{
    int ch [10];
    function1(ch);
    ...
}
```

//函数调用，实参是数组名

7.4.3 一维字符串数组

字符串是一维数组的一个普遍用法。在 C51 语言中,字符串被定义为一个以空字符结束的字符数组。其中空字符以“\0”来标识,它一般是不显示的。因此,在程序中声明字符数组时,必须声明一个比要存的字符串多一个字符的数组,最后一位用来存储空字符“\0”。

例如,假如要定义一个存放长度为 20 的字符串的数组 s,可以写成如下形式。

```
char s[21];
```

这样就给字符串末尾的空字符保留了空间。

在 C51 语言中,字符串并不是一种数据类型,但是可以使用字符串常量。字符串常量是由双引号括起来的一串字符。例如,下面两个短语均为字符串常量。

```
"hello everyone!","this is a test"
```

这里需要注意的是,不必向字符串的末尾加空字符,C51 语言编译程序会自动完成这一工作。在 C51 语言中提供了 string.h 库函数,其中提供了多个字符串操作函数供使用,示例如下:

- strlen(s1) 函数返回 s1 的长度。
- strcpy(s1,s2) 函数将 s2 复制到 s1。
- strcat(s1,s2) 函数将 s2 连接到 s1 的末尾。
- strcmp(s1,s2) 函数若 s1 与 s2 相等,返回值为 0;若 s1<s2,返回值小于 0;若 s1>s2,返回值大于 0。

下面给出字符串操作函数在程序中的应用,程序示例如下:

```
#include <stdio.h>                                //头文件
#include <string.h>                                //提供字符串操作的头文件

void main()                                        //主函数
{
    char a[10]={'a','b'};                          //声明字符串数组,并赋值
    char b[]="good";
    char s[30];
    printf("字符串 a 的长度为%d\n",strlen(a));      //输出字符串 a 的长度
    printf("字符串 b 的长度为%d\n",strlen(b));      //输出字符串 b 的长度
    strcat(a,b);                                    //将字符串 b 追加到 a 的末尾
    printf("a=%s,b=%s\n",a,b);                      //输出
    strcpy(s,a);                                    //将字符串 a 复制到字符串 s
    printf("s=%s\n",s);                              //输出字符串 s
}
```

该程序可以在 Keil μ Vision3 编译环境中执行,运行的结果如下:

```
字符串 a 的长度为 2
字符串 b 的长度为 4
a=abgood,b=good
s=abgood
```

在本程序中,首先定义了字符型数组 a、b 和 s,其中 a 和 b 在声明的时候已经赋值,然后分别调用了 strlen()函数、strcat()函数和 strcpy()函数进行字符串的操作。这些字符串函数位于 string.h 头文件中,因此必须加入该头文件。

在程序中,使用 strlen()函数求字符数组 a 和 b 的长度,因为 a 和 b 分别已赋 2 和 4 个字符,所以其长度分别为 2 和 4。使用 strcat()函数字符串 b 追加到 a 的末尾,此时字符串 a 将变为 a 和 b 的结合,即“abgood”,这也是为什么声明 a 时需要多声明其长度的原因。使用 strcpy()函

数，将字符数组 a 复制到 s 中，为 s 初始化，这样 s 和 a 的内容便相同。
头文件 string.h 中还有其他一些有用的函数，用户可以根据需要使用，这里不再一一介绍。

7.5 二维数组

二维数组是包含两个下标标号的数组。C51 语言允许使用多维数组，最简单的多维数组是二维数组。因此，这里首先介绍二维数组的声明和使用。

7.5.1 二维数组声明

实际上，二维数组可以看成是以一维数组为元素构成的数组，二维数组的一般声明形式如下：
类型说明符 数组名 [常量表达式 1][常量表达式 2]；

其中有两个方括号，“常量表达式 1”和“常量表达式 2”分别规定了该二维数组的行数和列数，整个数组元素的个数等于行数和列数的乘积。同一维数组一样，在声明二维数组时不能使用变量来表示数组的行数和列数，因为 C51 语言中不支持动态分配数组大小。二维数组的声明示例如下：

```
int ch[10][20];
```

该语句声明 ch 为（10，20）的二维整型数组，其有 10 行 20 列，共 200 个元素。

和大多数其他计算机语言不同的是，C51 不使用逗号区分下标，而是用方括号将各维下标括起。另外，数组的二维下标均从 0 计算。例如，要存取数组 ch 中下标为（3，5）的元素可以写成如下形式。

```
ch[3][5]
```

对二维数组赋值也需要和一维数组一样逐个元素赋值。示例如下：

```
#include <stdio.h>                                //头文件

void main()                                       //主函数
{
    int t,i,num[3][4];                           //定义整型二维数组
    for (t=0; t<3; ++t)                          //循环为其中的元素赋值
        for (i=0;i<4;++i)
            num[t][i]=(t*4)+i+1;
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序执行完毕后，数组 num 的数组元素值，如表 7-2 所示。

表 7-2 num 数组的数组元素的值

num	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

从表 7-2 中可以看出，二维数组以行列矩阵的形式存储。第一个下标代表行，第二个下标代表列，这意味着按照在内存中的实际存储顺序访问数组元素时，右边的下标比左边的下标变化快一些。C51 语言中二维数组的第一下标可以认为是行的指针。二维数组在单片机内存中的存放情形，如图 7-2 所示。

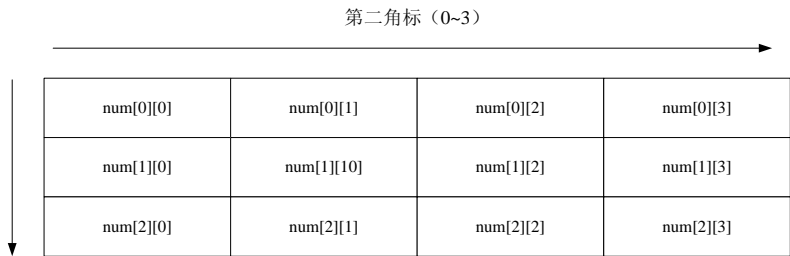


图 7-2 二维数组在内存中的存放情形

一个二维数组一旦被声明，所有的数组元素都将在单片机中分配相应的存储空间。二维数组计算所需内存总字节数的公式为：总字节数=行数×列数×类型字节数。

例如，二维数组 `d[10][5]`，其被定义为双字节整型，大小为 (10，5)。存放该数组需要占用单片机 $10 \times 5 \times 2 = 100$ 字节的存储空间。

7.5.2 二维数组初始化

在 C51 语言中，二维数组的初始化可以有如下三种方式。

1. 初始化方法 1

逐行给出数组元素值，示例如下：

```
int sum[2][5]={
    {5,4,3,2,1},           //第一行
    {-6,-4,6,3,0}};       //第二行
```


本例中，在声明整型二维数组 `sum` 时就对其进行了逐行初始化。数组 `sum` 有两行 5 列，首先用第一个花括号 “{5,4,3,2,1}” 为数组的第一行赋值，然后用第二个花括号 “{-6,-4,6,3,0}” 为数组的第二行赋值。每个花括号内都有 5 个数，正好赋值 5 个数组元素。

2. 初始化方法 2

按顺序逐个给出数组元素，示例如下：

```
int sum[2][5]={5,4,3,2,1,-6,-4,6,3,0};
```

本例中，在声明整型二维数组 `sum` 时就对其进行了连续初始化。这里将所有的元素值放在一个花括号中，并用逗号分隔。数组元素的分配顺序按照逐行逐列的方式，即先是第一行的列数从小到大，然后第二行的列数从小到大，依次直至整个数组赋值完毕。

 **说明：**一般来说，推荐使用第一种逐行赋值的方式，这样每一行什么元素值都一目了然，便于程序的阅读及后续的调试工作。

3. 初始化方法 3

二维数组也可以进行局部初始化，示例如下：

```
int num[4][4]={{1,2,3,4},{-2,0,-1,3},{7,10,-9,6}};
```

上面的语句只对数组的前三行进行赋值，而最后一行 `num[3][0]`、`num[3][1]`、`num[3][2]` 和 `num[3][3]` 的值均自动赋值为 0。

局部初始化还可以对列元素进行单独赋值，示例如下：

```
int num[4][4]={{1},{-2},{0},{3}};
```

上面的语句相当于只对二维数组 `num` 的第一列赋值，即 `num[0][0]=1`、`num[1][0]=-2`、

51 单片机开发与应用技术详解

num[0][0]=0 和 num[0][0]=3，而其余元素均自动赋值为 0。

下面举一个完整的例子来讲解二维数组在程序中的赋值以及使用，程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int t,i,ch[3][4];                             //定义整型变量和二维整型数组
    for (t=0; t<3; ++t)                           //两重循环为 ch 赋值
        for (i=0;i<4;++i)
            ch[t][i]=(t*4)+i+1;

    for (t=0; t<3; ++t)                           //循环，用于判断输出
        for (i=0;i<4;++i)
        {
            if(ch[t][i]%3==0)                     //判断元素值是否能被 3 整除
                printf("ch[%d][%d]=%d\n",t,i,ch[t][i]); //如果能够整除，则输出该元素
        }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
ch[0][2]=3
ch[1][1]=6
ch[2][0]=9
ch[2][3]=12
```

在本程序中，首先采用二重循环为二维数组 ch 循环赋值，然后再采用二重循环对每一个数组元素进行判断，如果其值是 3 的倍数，则输出结果。

7.5.3 二维字符串数组

二维字符串数组是二维数组的特殊形式，其元素值为字符。二维字符串数组的定义形式如下：

```
char 数组名[常量表达式 1][常量表达式 2];
```

其中，左下标决定字符串的个数，右下标说明字符串的最大长度。C51 语言程序设计中经常要用到字符串数组，二维字符串数组和一维字符串数组类似。二维字符串数组的声明示例如下：

```
char str[3][80];
```

该语句声明了一个二维字符串数组，可存放三个字符串，单个字符串的最大长度为 80 个字符。

在 C51 语言的程序设计中，二维字符串数组符合一维字符串数组的使用规则。这里仅举一个具体的例子程序，来讲解二维字符串数组的应用。程序示例如下：

```
#include <stdio.h>                                //头文件
#include <string.h>

void main()                                        //主函数
{
    char str[3][11]={"He looks ","like ","a good man"}; //声明并初始化字符串数组
    char str2[2][4]={{'a','b','c','\0'},{'A','B','C','\0'}}; //声明并初始化字符串数组
    //char str2[][]={"Good","morning!"};             //错误的声明方式，没有指定数组大小
    int i,j;                                       //定义整型变量

    for(i=0;i<2;i++)                             //循环逐个输出 str2 中的元素值
        for(j=0;j<4;j++)
```

```
printf("%c",str2[i][j]);

printf("\n");           //输出换行符
printf("%s\n",str);      //逐行输出字符串 str 的第一行
printf("%s%s%s\n",str,str+1,str+2); //逐行输出字符串 str 的每一行
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
abc ABC
He looks
He looks like a good man
```

在本程序中，采用了两种字符串数组初始化的方法。第一种是逐行赋值，特点是简单明了，数组 `str` 的赋值便是采用的这种；第二种方法是逐个赋值，`str2` 的赋值便是采用的这种方式，比较麻烦，不推荐使用。另外，一维字符串数组，可以在初始化时不指定大小，而采用自动分配大小；而二维字符串数组，则必须在声明时指定其大小。

在本程序中采用了两种方法访问二维字符串数组，即逐个访问和逐行访问。逐个访问需要指明该数组的行和列，而逐行访问比较简单，访问十分方便。

7.6 多维数组

多维数组是指数组的维数大于 2 的数组。在 C51 语言允许使用大于二维的数组，但维数的限制由具体的编译程序来决定的。多维数组的一般说明形式如下：

```
类型说明符 数组名[常量表达式 1] [常量表达式 2]... [常量表达式 n]
```

典型的多维数组的声明，示例如下：

```
int num[3][4][5][6];           //定义 4 维整型数组
char ch[6][7][8];              //定义 3 维字符型数组
```

多维数组占用字节数的计算同二维数组一样。例如，大小为（10，3，9，4）的四维字符数组需要 $10 \times 3 \times 9 \times 4$ 即 1080 字节的存储空间。如果这个数组是双字节整型的，则需要 2160 字节。

当多维数组定义之后，所有的数组元素都将分配到相应的地址空间。多维数组的存储量随着维数的增加呈指数增长，占用大量内存，因此在实际的程序设计中尽量少使用。

多维数组和二维数组满足相同的初始化和访问规则，因为其使用的比较少，这里就不做具体介绍了。关于多维数组，需要指出的一点是，单片机的 CPU 要花大量时间计算数组下标，这意味着存放和读取多维数组中的元素要比一维数组花更多的时间。

7.7 小结

本章首先介绍了 C51 语言中的数组的使用，主要包括数组类型的说明、数组元素的表示及初始化赋值。接着，本章分别对一维数组、二维数组及多维数组的使用进行了详细的讲解。数组是重要数据结构，广泛应用于 C51 语言的程序设计中。因此，熟练掌握本章内容是学习 C51 语言的基础。

第 8 章 指针

指针是变量在计算机或单片机内所占有的存储区域的地址。C51 语言中广泛使用的指针概念是从 C 语言中继承下来的，利用指针变量不但可以操作各种基本的数据类型和数组等复合数据结构，而且能使 C51 语言像汇编语言一样，具有处理单片机内存地址的能力。

在 C51 语言程序中，访问或修改变量可以通过两种方式来实现。一种是直接访问或修改这块区域的内容，另一种是先求出变量的地址，然后再通过地址对该变量的值进行访问，这就是本章所要讲解的指针及指针变量。

8.1 地址、指针和指针变量的概念

地址是单片机内存单元的编号。其中内存单元是单片机存储器中的最小存储单位，通常一个字节称为一个内存单元。当为内存单元逐个编号后，便可以根据一个内存单元的编号准确地找到该内存单元及其中的数据。

指针实际上也是一个地址，其表示的是一个变量在内存中的首地址。单片机中的数据都是存放在存储器中的，不同的数据类型所占用的内存单元数和它的大小有关。在 C51 语言中，根据数据类型或数据结构的不同，一个变量往往都占有一组连续的内存单元，单用“地址”这个概念并不能很好地描述这个变量，从而引进了“指针”的概念。指针是指向一个数据结构的首地址，因而概念更明确，寻址更方便。

指针变量是用来存放指针的一种变量类型。在 C51 语言中，一个指针变量的值就是某个内存单元的地址。在 C51 语言程序中定义指针的目的，是为了通过指针访问内存单元。

通过指针变量可以访问变量的地址及其他数据结构的地址，如数组、函数的首地址等。在 C51 语言程序中，数组或函数都是连续存放的。因此，通过访问指针变量取得了数组或函数的首地址，也就找到了该数组或函数。因此在程序中出现数组和函数的地方，用一个指针变量来表示和读写，就可以使程序精练、高效很多。

8.2 指针变量的声明

指针变量的声明与一般变量的声明相似，声明指针变量的一般形式如下：

类型标识符 *指针名 1, *指针名 2, ...;

其中前面的“类型标识符”表示该指针变量所指向的变量的类型，即指针变量的类型。典型的指针变量的声明示例如下：

```
int      *p1;           //p1 是指向整型变量的指针变量
char     *p2;           //p2 是指向字符变量的指针变量
float    *p3;           //p3 是指向浮点变量的指针变量
```

这里需要指出的是，一个指针变量只能指向同类型的变量，否则会出错。

8.3 指针变量的赋值

指针变量被声明后可以指向任何同类型的变量。如果定义指针变量时不进行初始化，编译系统不能确定它具体的指向，此时该指针变量被初始化为 `NULL`，它的值为 0。

这里需要强调的是，未经赋值的指针变量不能在程序中使用，否则将由指针指向错误导致整个程序的混乱。另外，指针变量的赋值只能赋予地址，决不能赋予任何其他数据，否则将引起错误。

在 C51 语言中，程序员不知道变量的具体地址，变量的首地址是由编译系统分配的。为此，C51 语言中提供了地址运算符“&”，可以获取变量的首地址。其一般形式如下：

`&变量名`

其中“&”为取地址运算符，变量名为预先声明的变量。例如，`&a` 表示变量 `a` 的首地址，`&b` 表示变量 `b` 的首地址。

在 C51 语言中，指针变量的赋值可以有下面几节介绍的几种方式。

8.3.1 初始化赋值

假设有指向整型变量的指针变量 `p` 和整型变量 `a`，如要把整型变量 `a` 的地址赋予 `p`，可以采用初始化赋值，示例如下：

```
int a;
int *p=&a;           //初始化赋值，&a 表示取变量 a 的首地址
```

8.3.2 取地址赋值

假设有指向整型变量的指针变量 `p` 和整型变量 `a`，如要把整型变量 `a` 的地址赋予 `p`，可以采用取地址赋值，示例如下：

```
int a;
int *p;
p=&a;                 //利用&a 获得变量 a 的首地址，然后赋值给 p
```

❗注意：不允许把一个数值赋予指针变量，故 `int *p;p=200` 的赋值语句是错误的。

被赋值的指针变量前不能再加“*”说明符，如写为 `*p=&a` 是错误的，指针变量初始化时除外，示例如下：

```
int a,*p=&a;
```

本例中，是用 `&a` 对指针变量 `p` 初始化，而不是对 `*p` 初始化。

8.3.3 指针之间赋值

把一个指针变量的值直接赋予相同类型变量的另一个指针变量。示例如下：

```
int a=5,b=6;           //定义变量和初始化
int*p1=&a,*p2=&b;       //定义指针变量和初始化
p2=p1;                 //把 a 的地址赋予指针变量 p2
*p2=*p1;               //把 p1 指向的内容赋给 p2 所指的区域
```

本例中，由于指针变量 `p1`、`p2` 均指向整型变量，因此它们之间可以互相赋值。赋值操作时的指针指向，如图 8-1 和图 8-2 所示。

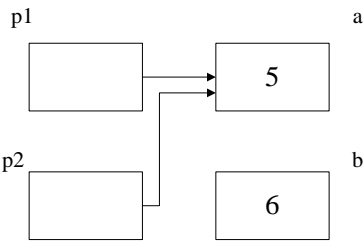


图 8-1 p2=p1 的情况

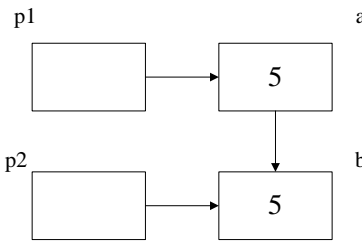


图 8-2 *p2=*p1 的情况

8.3.4 数组赋值

把数组的首地址赋予指向数组的指针变量。示例如下：

```
int a[5],*p;           //声明数组以及指针变量
p=a;                   //数组名表示数组的首地址，故可赋予指向数组的指针变量
```

由于数组在内存中是连续存放的，数组的首地址即数组中第一个元素的地址，因此也可写为如下形式。

```
p=&a[0];                //数组第一个元素的地址也是整个数组的首地址
```

同样，采取初始化赋值的方法为指针变量进行赋值，示例如下：

```
int a[5],*p=a;          //声明数组及指针变量并进行初始化赋值
```

8.3.5 字符串赋值

把字符串的首地址赋予指向字符类型的指针变量。示例如下：

```
char *p;                //声明字符型指针变量
p="This is a test";     //赋值
```

也可以采用初始化赋值的方法写为如下形式。

```
char *p="This is a test";
```

这里应注意的是，该语句并不是把整个字符串装入指针变量，而是把存放该字符串的字符数组的首地址装入指针变量。

8.3.6 函数入口赋值

把函数的入口地址赋予指向函数的指针变量。示例如下：

```
int (*pfun)();
pfun=fun;                //fun 为函数名
```

在 C51 语言中，指针变量若不带初始化项，其将被初始化为 NULL，其值为 0。这一点与一般变量类似。当指针变量的值为零时，表明该指针不指向任何有效数据，此时称为“空指针”。空指针常用于调用一个带有返回指针的函数时，如果返回值为 NULL，则指示该函数调用中出现某些错误情况。

8.4 指针变量的引用

指针变量的引用即通过指针变量来间接访问其所指向地址中的内容。引用指针变量需要首先声明该指针变量。下面介绍与指针变量的引用有关的两个运算符。

8.4.1 取地址运算符“&”

取地址运算符“&”用于取变量的地址。取地址运算符是单目运算符，符合自右至左的结

合性。在上一节中，已经使用了“&”运算符。对指针变量的引用形式如下：

&变量名

其中，在“&”运算符之后跟的变量必须是所要取址的变量。该语句的含义是获得变量在单片机实际内存中的地址。

8.4.2 取内容运算符“*”

取内容运算符“*”用来表示指针变量所指向内存中的数据内容。取内容运算符也是单目运算符，符合自右至左的结合性。对指针变量的引用形式如下：

*指针变量名

其中，在“*”运算符之后跟的变量必须是指针变量。该语句的含义是获得指针变量所指向的单片机内存地址中的数值。

这里需要注意的是，取内容运算符“*”和指针变量说明中指针说明符“*”的区别。在指针变量说明中，“*”是类型说明符，表示其后是指针类型的变量。而表达式中出现的“*”则是一个运算符，用来表示指针变量所指向的地址中的数据值。示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int x=1, y, *p=&x;                            //指针变量 p 指向整数 x，则 *p 等价于 x
    y=*p+21;                                       //表示把 x 的内容加 21 并赋给 y，即 y=x+21
    printf("y=%d\n", y);                          //输出
    y=++*p;                                       // *p 的内容加上 1 之后赋给 y，即 y=x+1
    printf("y=%d\n", y);                          //输出
    y=*p++;                                       //相当于 y=x; x++;
    printf("y=%d\n", y);                          //输出
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
y=22
y=2
y=1
```

与直接访问一个变量相比，通过指针访问它所指向的变量显得不直观，因为通过指针要访问哪一个变量，取决于指针的值（即指向），也就是以间接访问的形式进行的。使用时需要认真分析指针变量的指向，下面再举一个例子，程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int i=1, j=2, *p2, *p1;                       //声明变量并赋值
    p1=&i;                                         //指针变量 p1 赋值
    p2=&j;                                         //指针变量 p2 赋值
    *p2=*p1;                                       //“*p2=*p1”实际上就是“j=i”
    p1=p2;                                         //使 p1、p2 指向同一对象 j
}
```

在本例中，与句“*p2=*p1”实际上就是“j=i”，而“p1=p2”使 p1、p2 指向同一对象 j。此时 *p1 等价于 j，而不是 i。

在实际的程序设计中，由于指针是一个变量，因而可以通过改变指针变量的指向，来间接访问不同的变量，这样可以使程序代码更为简洁、灵活。

8.5 指针变量的运算

由于指针变量的特殊性，其可以进行有限的某些运算，例如赋值运算、部分算术运算及关系运算。在使用指针变量进行运算时，需要注意如下两点。

- 只有指向同一数组的两个指针变量之间才能进行运算，否则运算无意义。
- 指针变量的加减运算只能对数组指针变量进行，对指向其他类型变量的指针变量做加减运算是毫无意义的。

对于赋值运算，前面已有介绍。下面介绍指针变量在关系运算和算术运算中的使用。

8.5.1 关系运算

对于指向同一数组的两指针变量进行关系运算，可用来表示它们所指向数组元素之间的关系。例如两个指针变量 p 和 q 指向同一数组，则 $<$ 、 $>$ 、 $>=$ 、 $<=$ 、 $=$ 等关系运算符都能正常使用。

- 若 $p==q$ 为真，则表示 p 和 q 指向同一数组元素；
- 若 $p<q$ 为真，则表示 p 处于低地址位置， q 处于高地址位置；
- 若 $p>q$ 为真，则表示 p 处于高地址位置， q 处于低地址位置。

另外，指针变量未赋值时，系统自动为其赋值为 `NULL`，既空指针，其值为 0，不指向任何变量。因此，指针变量还可以与 0 比较。设 p 为指针变量，则 $p==0$ 为真表示 p 是空指针，它不指向任何变量； $p!=0$ 为真表示 p 不是空指针。

8.5.2 算术运算

指针变量和整数可进行简单的加、减运算。假设 p 是指向数组 a 的指针变量，开始时指向数组的某个元素，设 n 为一个正整数，则如下运算是合法的。

- $p+n$ ，指针变量指向的位置向后移动 n 个位置；
- $p-n$ ，指针变量指向的位置向前移动 n 个位置；
- $p++$ ，指针变量指向的位置向后移动 1 个位置；
- $++p$ ，首先取指针变量的当前位置，然后将指针变量指向的位置向后移动 1 个位置；
- $p--$ ，指针变量指向的位置向前移动 1 个位置；
- $--p$ ，首先取指针变量的当前位置，然后将指针变量指向的位置向前移动 1 个位置。

总之，指针变量加或减一个整数 n 的意义是把指针指向的当前位置（指向某数组元素）向前或向后移动 n 个位置。

指向数组的指针变量向前或向后移动一个位置和地址加 1 或减 1 是不同的。因为数组可以有不同的类型，各种类型的数组元素所占的字节长度是不同的。因此，如果指针变量加 1，即向后移动 1 个位置表示指针变量指向下一个数据元素的首地址，而不是在原地址基础上加 1。

从编译系统角度来看，不论指针变量指向何种数据类型，指针和整数进行加、减运算时，编译程序总根据所指对象的数据长度对 n 放大，在一般编译系统中，`char` 放大因子为 1，`int`、`short` 放大因子为 2，`long` 和 `float` 放大因子为 4，`double` 放大因子为 8。对于下面讲到的结构或联合，也仍然遵守这一原则。

两个指针变量在一定条件下，可进行减法运算。两指针变量相减所得之差是两个指针所指数组元素之间相差的元素个数。实际上是两个指针值（地址）减之差再除以该数组元素的长度（字节数）。

例如， $p1$ 和 $p2$ 是指向同一浮点数组的两个指针变量，设 $p1$ 的值为 2010H， $p2$ 的值为 2000H，而浮点数组每个元素占 4 个字节，所以 $p1-p2$ 的结果为 $(2010H-2000H)/4=4$ ，表示 $p1$ 和 $p2$ 之间相差 4 个元素。其相减的结果遵守对象类型的字节长度进行缩小的规则。

!注意：两个指针变量之间不能进行加法运算，而且这样做毫无实际意义。

8.6 数组指针

数组指针是指向数组的指针变量。变量在内存中存放是有地址的，数组在内存存放也同样具有地址。对数组来说，数组名就是数组在内存存放的首地址。指针变量是用于存放变量的地址，可以指向变量，当然也可以指向数组的首地址或数组元素的地址，即指针变量可以指向数组或数组元素。对数组而言，数组和数组元素的引用，也同样可以使用指针变量来完成。下面分别介绍指向不同类型数组的指针变量。

8.6.1 指向一维数组的指针

将一个指针指向一维数组，需要首先定义一个整型数组和一个指向整型的指针变量。示例如下：

```
int a[10];           //声明数组
int *p;             //声明指针变量
p=&a[0];             //指针指向数组的第0号元素
```

在上面的语句中，语句 `p=&a[0]` 将指针变量 `p` 指向数组 `a` 中的第 0 号元素，即 `a[0]`。指针变量 `p` 中的数值数组元素 `a[0]` 的地址。另外，在 C51 语言中，数组名表示数组的第 0 个元素的地址，因此下面两个语句是等价的。

```
p=&a[0];
p=a;
```

由于数组元素在内存中是连续存放的，因此通过指针变量 `p` 及其有关运算可以间接访问数组中的任何一个元素。根据上一节介绍的指针变量的运算规则，下面用指针给出数组元素的地址和内容的几种表示形式。

- `a+i` 和 `p+i` 均表示数组元素 `a[i]` 的地址，即它们均指向数组第 `i` 个元素，即指向 `a[i]`；
- `*(p+i)` 和 `*(a+i)` 都表示 `p+i` 和 `a+i` 所指向元素的值，即为数组元素 `a[i]` 的值；
- C51 语言允许指针变量带下标，即指向数组元素的指针也可以表示成数组的形式，即如 `p[i]` 与 `*(p+i)` 等价。示例如下：

```
int a[10];           //声明数组
int *p;             //声明指针变量
p=a+7;              //指针变量赋初值
```

上面语句中 `p[2]` 就相当于 `*(p+2)`，由于 `p` 初始值指向 `a[7]`，所以 `p[2]` 就相当于 `a[9]`，而 `p[-2]` 就相当于 `*(p-2)`，它表示 `a[5]`。下面举一个完整的一维数组指针的例子，程序示例如下：

```
#include <stdio.h>           //头文件

void main()                 //主函数
{
    int i;                  //定义整型变量
    int ch[10]={0,1,2,3,4,5,6,7,8,9}; //声明并初始化整型数组 a
    int *p;                 //定义整型指针变量 p
    p=ch;                   //将数组 ch 的首地址赋值给 p
    for(i=0;i<10;i++)       //循环输出指针 p 所指向的内容
        printf("%d ",*(p+i));
    printf("\n");           //输出换行符
    for(i=0;i<10;i++)       //循环输出指针 p 所指向的内容
```

```
printf("%d ",p[i]);  
printf("\n");           //输出换行符  
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
0 1 2 3 4 5 6 7 8 9  
0 1 2 3 4 5 6 7 8 9
```

在本程序中，首先将整型数组 `ch` 的首地址赋值给整型指针变量 `p`，接着分别采用了两种方法输出 `p` 所指向的内存空间的值，即数组 `ch` 中元素的值。

8.6.2 指向二维数组的指针

将一个指针指向二维数组，需要首先定义一个二维数组，示例如下：

```
char a[3][4]={{'a','b','c','d'},{'e','f','g','h'},{'i','j','k','l'}};
```

这里声明并初始化了一个二维字符型数组，共有 3 行 4 列 12 个元素。在 C51 语言中，这个二维数组 `a` 可以看成是由三个一维数组元素组成，即 `a[0]`、`a[1]`、`a[2]`。其中每个一维数组元素又是一个包含有 4 个元素一维数组，分别如下：

- `a[0]` 所代表的一维数组所包含的 4 个元素为 `a[0][0]`、`a[0][1]`、`a[0][2]`、`a[0][3]`；
- `a[1]` 所代表的一维数组所包含的 4 个元素为 `a[1][0]`、`a[1][1]`、`a[1][2]`、`a[1][3]`；
- `a[2]` 所代表的一维数组所包含的 4 个元素为 `a[2][0]`、`a[2][1]`、`a[2][2]`、`a[2][3]`。

该二维数组的结构，如图 8-3 所示。

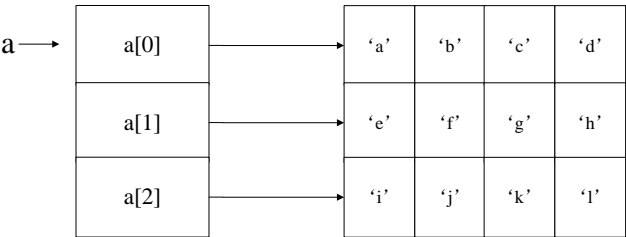


图 8-3 二维数组的结构

在 C51 语言中，数组名 `a` 代表二维数组的首地址，也就是二维数组 `a` 第 0 行的首地址。按照前面介绍的指针变量的运算规则，`a+1` 就代表第 1 行的首地址，`a+2` 就代表第 2 行的首地址。

例如，如果二维数组 `a` 在内存中的首地址为 `0x2000`，由于每一行有 4 个单字节的字符型元素，所以 `a+1` 的地址为 `0x2004`，`a+2` 也就为 `0x2008`，如图 8-4 所示。

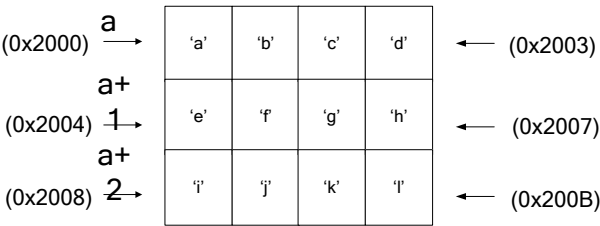


图 8-4 二维数组的指针

二维数组同样需要根据元素的类型，来对地址进行适当的放大。假如 `a[3][4]` 声明为二维整型数组，此时每个数组元素占两个字节的存储单元。因此，假如 `a` 的首地址为 `0x2000`，则 `a+1` 的地址为 `0x2008`，`a+2` 的地址为 `0x2010`。

在 C51 语言中，`a[0]`、`a[1]`、`a[2]` 可以看成是一维数组名，它们分别代表所对应的数组的首地址。此时，根据前面的介绍，满足如下的运算规则。

- `a[0]` 代表第 0 行第 0 列元素的地址，即 `&a[0][0]`；`a[1]` 是第 1 行第 0 列元素的地址，即 `&a[1][0]`。

➤ `a[0]+1` 即代表第 0 行第 1 列元素的地址，即 `&a[0][1]`。一般来说 `a[i]+j` 代表第 `i` 行第 `j` 列元素的地址，即 `&a[i][j]`。

在二维数组中，数组名类似于指针变量。因此，可用指针的形式来表示各元素的地址，示例如下：

```
a[0]=&a[0][0]=*(a+0);
a[i]+j=&a[i][j]=*(a+i)+j;
```

对于二维数组元素，以下的几种表示是等价的。

```
a[i][j]=*(a[i]+j)=*(*(a+i)+j)= (*(a+i))[j];
```

引入数组指针的概念后，二维数组的地址和元素的表示方法很灵活，使用时需要注意。下面举一个例子说明它们的使用方法，程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                         //主函数
{
    int ch[3][4]={{1,1,3,3},{5,6,7,8},{9,10,1,2}}; //定义并初始化二维整型数组 ch
    int *p;                                         //定义整型指针变量
    int i,j;                                        //定义整型变量
    for(i=0;i<3;i++)                               //循环输出 ch 中的元素方法一
        for(j=0;j<4;j++)
            printf("%d ",ch[i][j]);
    printf("\n");                                   //输出换行符
    for(i=0;i<3;i++)                               //循环输出 ch 中的元素方法二
        for(j=0;j<4;j++)
            printf("%d ",*(ch+i+j));
    printf("\n");                                   //输出换行符
    for(p=ch[0];p<a[0]+12;p++)                     //循环输出 ch 中的元素方法三
        printf("%d ",*p);
    printf("\n");                                   //输出换行符
}
```

程序中分别用三种方法输出二维整型数组 `ch` 中的元素。该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
1 1 3 3 5 6 7 8 9 10 1 2
1 1 3 3 5 6 7 8 9 10 1 2
1 1 3 3 5 6 7 8 9 10 1 2
```

再举一个例子，示例如下：

```
#include <stdio.h>                                //头文件


void main()                                         //主程序
{
    int ch[3][4]={{1,1,3,3},{5,6,7,8},{9,10,1,2}}; //定义并初始化二维整型数组 ch
    printf("ch=%d\n",ch);                           //输出 ch 的值
    printf("*ch=%d\n",*ch);                         //输出 *ch 的值
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
a=1000
*a=1000
```

从上面结果中可以看出，程序输出 `a` 和 `*a` 的结果是相同的。这是因为，在 C51 语言程序中，二维数组是由三个一维数组元素 `a[0]`、`a[1]`、`a[2]` 组成。因此 `a` 表示数组第 0 行的地址，而 `*a`

表示 `a[0]`，`a[0]` 是一维数组名，还是一个地址，即是二维数组第 0 行第 0 列元素的地址。因此，它们输出的都是相同的地址值。

 **说明** 这段程序在不同的编译环境和硬件系统中可以有不同的输出结果，这是正常的，但是 `a` 和 `*a` 的输出结果必定是相同的。

8.6.3 指向一个由 n 个元素所组成的数组指针

在 C51 语言中，还引入了一个指向由 n 个元素构成的数组指针。这样可以方便二维数组的处理，其定义格式如下：

类型标识符 (*指针名) [n];

其中，类型标识符表示数组指针的类型，指针名即数组指针的变量名。示例如下：

```
int ch[3][4];           //定义二维数组
int (*p)[4];           //定义指针
p=ch;                  //指针赋值
```

其中，指针 `p` 是指向一个由 4 个元素所组成的整型数组指针。这种数组的指针有一个特点，当整型指针指向一个整型数组的元素时，进行指针加 1 运算，其地址值实际增加了 8（放大因子为 $2 \times 4 = 8$ ），既相当于指向数组隔行的元素。例如，这里 `*p=ch[0][0]`，则下一个元素 `*p[1]=ch[1][0]`。

下面举一个具体的例子，介绍指向一个由 n 个元素所组成的数组指针的用法。程序示例如下：

```
#include <stdio.h>           //头文件

void main()                 //主函数
{
    int ch[3][4]={ {1,1,3,3}, {5,6,7,8}, {9,10,1,2} }; //定义并初始化二维整型数组 ch
    int (*p)[4];            //定义由三个元素组成的数组指针
    int i;                  //定义整型变量
    p=ch;                   //将指针 p 指向 ch
    for(i=0;i<3;i++)        //循环输出*p[i]的值
        printf("%d ",*p[i]);
    printf("\n");           //输出换行符
    for(i=0;i<3;i++)        //循环输出*(p[i]+1)的值
        printf("%d ",*(p[i]+1));
    printf("\n");           //输出换行符
    p=&ch[0][2];            //重新修改 p 的指向
    for(i=0;i<3;i++)        //循环输出*p[i]的值
        printf("%d ",*p[i]);
    printf("\n");           //输出换行符
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
1 5 9
1 6 10
3 7 1
```

8.6.4 指针和数组的关系

在 C51 语言中，指针和数组有着密切的关系，其使用十分灵活，在使用中需要注意如下两点。

- 任何能由数组和下标完成的操作，也都完全可以用指针和指针的偏移量来实现；

➤ 在程序中使用指针可使代码更灵活，也可以使程序执行得更快，并使生成的目标代码更小。

由于指针和数组的使用十分容易混淆，这里列出它们之间的一些典型关系，以加深读者的理解。对于一维数组及指针变量，声明示例如下：

```
int a[3];
int *p;
p=a;
```

具有如下的几种等价操作。

```
a+i=p+i; (i=0,1,2) //关于地址的运算
a[i]=*(a+i)=p[i]=*(p+i); (i=0,1,2) //关于元素的运算
```

对于二维数组以及指针变量，声明示例如下：

```
int a[3][4];
int *p;
p=a;
```

具有如下的几种等价操作。

```
a=*a=a[0]=p; //关于地址的运算
a[i]=*(a+i)=*(p+i); (i=0,1,2) //关于地址的运算
&a[i][j]=a[i]+j=*(a+i)+j=*(p+i)+j; (i=0,1,2;j=0,1,2,3) //关于地址的运算
a[i][j]=*(a[i]+j)=*(*(a+i)+j)=*(*(p+i)+j)= (*(p+i))[j];
(i=0,1,2;j=0,1,2,3) //关于元素的运算
```

8.7 字符指针

字符指针是指向字符类型的指针变量。由于在 C51 语言中没有字符串变量，只有字符串常量。采用字符指针可以对字符串常量进行操作，即用字符指针指向字符串，然后通过字符指针来访问字符串的存储区域。

在 C51 语言中，字符串常量是由双引号括起来的一个字符序列，示例如下：

```
"Good morning"
```

该字符串由 13 个字符序列组成，其中 12 个字母和最后一位空字符“\0”。在程序执行过程中，C51 语言程序自动对字符串常量分配一个连续的存储空间，其中按顺序存储字符，最后，系统会自动在字符串的末尾加上空字符“\0”。这段连续的存储单元是静态的，在程序运行过程中始终被占用。

前面章节介绍的操作一个字符串常量的方法，是将字符串常量存放在一个字符数组之中。示例如下：

```
char str[]="Good morning ";
```

数组 str 共有 13 个元素所组成，其中 str[12]中的内容是空字符“\0”。

当引入字符指针后，便可以使用字符指针来对字符串进行操作。例如，定义一个字符指针 pstr，代码如下：

```
char *pstr;
```

让该字符指针指向一个字符串，可以采用如下的语句。

```
pstr="Good morning";
```

该语句执行后，字符型指针 pstr 便指向字符串常量中的首字符“G”，如图 8-5 所示。

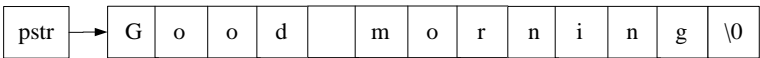


图 8-5 字符指针

下面通过一个具体的例子，来讲解一下字符型指针的使用。程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    char str[]="Good morning";                    //定义并初始化字符数组 str
    char *pstr;                                    //定义字符型指针
    pstr="Good morning";                          //将字符型指针指向字符串
    printf("str=%s\n",str);                        //输出字符串 str 的值
    printf("*pstr=%c,pstr[0]=%c\n",*pstr,pstr[0]); //输出地址 pstr 中的内容
    printf("pstr[3]=%c, *(pstr+3)=%c\n",pstr[3], *(pstr+3)); //输出 pstr[3] 中的内容
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str=Good morning
*pstr=G, pstr[0]=G
pstr[3]=d, *(pstr+3i)=d
```

通过本程序，可以看出利用字符型指针 pstr 访问字符串常量，*pstr 是字符串的第一个字符，也可以用 pstr[0]来表示。pstr[i]和*(pstr+i)都表示字符串中的第 i 个元素。


注意：由于常量的值不可改变，因此不能通过指针来修改字符串常量。

8.8 指针数组

指针数组是同一数据类型的指针构成的一个数组。其中，指针数组中的每个元素都是指针变量。指针数组的定义格式如下：

```
类型标识符 *数组名[整型常量表达式];
```

其中，类型标识符是指针数组的类型，“[]”内的整型常量表达式为指针数组的大小。根据数组的定义，指针数组中每个元素都为指向同一数据类型的指针。

 **说明：**这里需要注意指针数组与数组指针的区别。

典型的指针数组的声明示例如下：

```
char *ch[10];
```

该语句定义了一个指针数组 ch，数组中的每个元素都是指向字符型变量的指针。该数组由 10 个元素组成，即 ch[0],ch[1],ch[2],...,ch[9]，它们均为指针变量。

指针数组名为 ch，其满足和一般的数组一样运算规则，示例如下：

- ch 为指针数组元素的首地址；
- ch+i 为指针数组的第 i 个元素的地址；
- *ch 等价于 ch[0]，为指针数组的第一个元素；
- *(ch+i)等价于 ch[i]，为数组的第 i 个元素。

引入指针数组，极大地方便了对字符串的处理。对于字符串数组，如果每个字符串的长度不一致，用二维数组来表示时，必须按最长的字符串来定义二维数组的长度。这样十分浪费单片机资源。例如，定义字符串数组 week，用来存储每个星期的英文名，采用数组方式的声明方式如下：

```
char week[7][10]={"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday",
"Sunday"};
```

其中，虽然有些英文单词比较少，但必须按最长的字符串“Wednesday”来定义数组长度，即必须定义为 week[7][10]。如果采用指针数组，则可以解决这个问题，其定义如下：

```
char *pweek[7]={"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday",
"Sunday"};
```

该语句只定义了一个一维字符型指针数组。赋值操作将存放字符串的首地址赋给指针数组的对应元素。数组的每个元素对应于一个字符串，pweek[0]指向“Monday”、pweek[1]指向“Tuesday”等。这样便不会浪费单片机紧缺的内存单元，使字符串的操作更加方便。

下面举一个具体的例子，介绍指针数组的具体使用方法。程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int k;
    char *pweek[7]={ "Monday", "Tuesday", "Wednesday",      //定义并初始化指针数组
                    "Thursday", "Friday", "Saturday", "Sunday"};
    for(k=0; k<7; k++)                             //循环输出指针数组指向的字符串
        printf("pweek[%d]=%s\n", k, pweek[k]);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
pweek[0]=Monday
pweek[1]=Tuesday
pweek[2]=Wednesday
pweek[3]=Thursday
pweek[4]=Friday
pweek[5]=Saturday
pweek[6]=Sunday
```

程序中，首先声明并初始化了指针数组，用来存放英文字符串，然后通过循环语句将指针数组的每个元素对应的字符串输出。

8.9 小结

本章首先介绍了 C51 语言中指针的使用，主要包括指针的概念、指针变量的声明和赋值。接着，本章详细讲解了指针变量的引用及运算规则。最后，本章还对几种特殊的指针进行了详细的介绍，主要包括数组指针、字符指针及指针数组。指针的概念来源于 C 语言，在程序设计中灵活使用指针可以优化程序设计。因此，熟练掌握本章内容是学习 C51 语言的基础，同时对读者以后的 C51 语言程序设计工作会大有帮助。

第9章 结构

在实际的程序设计中，经常需要处理具有不同数据类型的一组数据。例如，在学生成绩统计表中，应该包含如下几项：姓名（字符型）、学号（整型或字符型）、年龄（整型）、性别（字符型）和成绩（整型或浮点型）。由于数组中各元素的类型和长度都必须一致，因此不能用一个数组来存放这一组数据。这种情况下，可以使用“结构”，这是一种构造数据类型，相当于其他高级语言中的记录类型。

9.1 结构的定义

结构是一种构造类型，是由数目固定、类型不同的若干有序成员集合而成的数据类。其中的每一个成员可以是一个基本数据类型，甚至可以是一个构造类型。由于结构不是基本的数据类型，因此在说明和使用之前必须先定义。结构的一般定义形式如下：

```
struct 结构名
{
    类型说明符 成员名;
    类型说明符 成员名;
    ...
};
```

其中，struct 为关键字，结构名即所要定义的结构，类型说明符为每个内部成员的类型。

这里需要注意，结构定义的最后一定要加上分号“;”。

在 C51 语言中，允许在定义结构的时候同时声明结构变量，其一般形式如下：

```
struct 结构名
{
    类型说明符 成员名;
    类型说明符 成员名;
    ...
} 结构变量;
```

其中，最后的结构变量即程序中所采用的变量名。示例如下：

```
struct student                                //声明结构
{
    char name[20];                            //姓名
    int num;                                  //学号
    int age;                                  //年龄
    char sex[2];                              //性别
    float score;                             //分数
}Bob;                                         //结构变量
```

本例中，定义了一个结构 `student` 的同时也声明了一个结构变量 `Bob`。该结构由 5 个内部成员组成，第 1 个为 `name`，字符数组；第 2 个成员为 `num`，整型变量；第 3 个成员为 `age`，整型变量；第 4 个成员为 `sex`，字符变量；第 5 个成员为 `score`，实型变量。

如果在定义结构的时候没有声明结构变量名，则可以在结构定义之后，单独进行结构变量的声明。示例如下：

```
student Alice,Bob;
```

凡定义为结构 `student` 的变量都由上述 5 个变量组成。

9.2 结构变量的定义

在 C51 语言中，声明结构变量可以采用如下的三种方法。下面均以前面定义的学生成绩统计结构 `student` 为例来加以说明。

9.2.1 先定义结构，再定义结构变量

示例如下：

```
struct student
{
char name[20];
int num;
int age;
char sex[2];
float score;
};
struct student Alice,Bob;
```

其中，首先定义了结构 `student`，然后声明了结构变量 `Alice` 和 `Bob`。

在 C51 语言中，也可以用宏定义来使一个符号常量来表示一个结构类型，示例如下：

```
#define STU struct student
STU
{
char name[20];
int num;
int age;
char sex[2];
float score;
};
STU boy1,boy2;
```

其中，首先宏定义 `STU` 为 `struct student`，然后对 `STU` 进行定义，最后使用 `STU` 对结构变量 `Alice` 和 `Bob` 进行声明。

9.2.2 在定义结构的同时，定义结构变量

在 C51 语言中，在定义结构的同时可以声明多个结构变量，示例如下：

```
struct student
{
char name[20];
int num;
int age;
char sex[2];
float score;
}Alice,Bob;
```

其中，在定义结构 `student` 的同时，声明了 `Alice` 和 `Bob` 两个结构变量。

9.2.3 直接说明结构变量

在 C51 语言中，还可以在程序中直接说明结构变量，而不显式地说明结构名称。示例如下：

```
struct
{
char name[20];
int num;
int age;
char sex[2];
float score;
}Alice,Bob;
```

其中，在定义结构的时候没有给出结构的名称，在程序中使用此处声明的两个结构变量 Alice 和 Bob，而不能再额外声明其他的结构变量。

第三种方法与第二种方法的区别在于第三种方法中省去了结构名，而直接给出结构变量。无论采用任何方法，在说明了变量 Alice 和 Bob 为 student 类型后，便可向这两个变量中的各个成员赋值并使用该变量了。

9.3 结构变量的使用

在 C51 语言中，表示结构变量成员的一般形式如下：

结构变量名.成员名

其中，结构变量名为声明的结构类型变量，成员名为结构中的成员。在 C51 语言中，允许具有相同类型的结构变量相互赋值，同时允许对结构变量中的成员进行赋值、修改等操作。以前面定义的结构变量 Alice 和 Bob 为例。

- Alice.num 即 Alice 的学号；
- Bob.sex 即 Bob 的性别。

下面举一个具体的例子，介绍结构变量在程序中的使用。程序示例如下：

```
#include <stdio.h>                                //头文件

struct student                                    //定义结构
{
char *name;                                       //姓名
int num;                                         //学号
int age;                                         //年龄
char *sex;                                       //性别
float score;                                     //分数
};

void main()                                       //主函数
{
struct student Bob;                             //声明结构变量
Bob.name="Bob ";                               //对每个成员进行赋值
Bob.num=210;
Bob.age=22;
Bob.sex="M";
Bob.score=91.5;

//输出结构变量的每个成员值
printf("%s %d %d %s %f\n",Bob.name,Bob.num,Bob.age,Bob.sex,Bob.score);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Bob 210 22 M 91.500000
```

程序中，首先定义 student 结构，然后在主程序中声明结构变量 Bob，并对其中的各个成员进行赋值，最后打印输出 Bob 的各个成员内容。

9.4 多重结构变量的赋值

多重结构变量是指结构变量的成员本身又是一个结构。对于这样的多重结构，必须逐级找到最低级的成员才能使用。示例如下：

```
#include <stdio.h>                                //头文件

struct kemu                                        //定义结构 kemu
{
    float Chinese;                                //语文成绩
    float English;                                //英语成绩
    float Math;                                    //数学成绩
};

struct student                                    //定义结构 student
{
    char *name;                                    //姓名
    int num;                                       //学号
    int age;                                       //年龄
    char *sex;                                    //性别
    struct kemu score;                            //成绩
};

void main()                                       //主函数
{
    struct student Bob;                          //声明结构变量
    Bob.name="Bob ";                             //对每个成员进行赋值
    Bob.num=210;
    Bob.age=22;
    Bob.sex="M";
    Bob.score.Chinese=90.5;
    Bob.score.English=87.5;
    Bob.score.Math=96.0;

    printf("%s %d %d %s %f %f %f \n",           //输出变量的成员值
        Bob.name, Bob.num, Bob.age, Bob.sex, Bob.score.Chinese, Bob.score.English,
        Bob.score.Math);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Bob 210 22 M 90.500000 87.500000 96.000000
```

程序中，首先定义结构 kemu 及结构 student。其中 student 为多重结构，最后一个成员 score 为 kemu 结构类型。在主程序中，首先声明结构变量 Bob，并对其中的各个成员进行赋值，最后打印输出 Bob 的各个成员内容。

9.5 结构变量的初始化

在 C51 语言中，结构变量声明的同时也可以进行初始化，其一般形式如下：


```
struct 结构名
{
    类型说明符 变量名;
    类型说明符 变量名;
    ...
}结构变量={值, 值, ...};
```

其中，最后的“{}”内的值便是结构变量的初始值。下面举例说明其使用方法，程序示例如下：

```
#include <stdio.h>                                     //头文件
struct kemu                                             //定义结构 kemu
{
    float Chinese;                                     //语文成绩
    float English;                                    //英语成绩
    float Math;                                        //数学成绩
};
struct student                                         //定义结构 student
{
    char *name;                                       //姓名
    int num;                                         //学号
    int age;                                         //年龄
    char *sex;                                       //性别
    struct kemu score;                               //成绩
}Bob={"Bob",210,22,"M",{90.5,87.5,96.0}};

void main()                                           //主函数
{
    printf("%s %d %d %s %f %f %f \n",               //输出变量的成员值
        Bob.name, Bob.num, Bob.age, Bob.sex, Bob.score.Chinese, Bob.score.English,
        Bob.score.Math);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Bob 210 22 M 90.500000 87.500000 96.000000
```

程序中，首先定义结构 kemu 及结构 student。其中 student 为多重结构，最后一个成员 score 为 kemu 结构类型。在定义 student 结构的同时，声明并初始化了结构变量 Bob。在主程序中，打印输出 Bob 的各个成员内容。

这里需要注意的是，不能在结构体内赋值。例如，如下的代码是错误的。

```
struct student
{
    char name[20]="Bob";
    int num=210;
    int age=22;
    char sex[2]="M";
    float score=97.5;
}Bob;
```

9.6 结构数组

结构数组是指数组中的每一个元素都是具有相同结构类型的结构变量。在实际应用中，经常用结构数组来表示具有相同数据结构的一个群体。如一个班的学生成绩档案，一个公司员工

的工资表等。

9.6.1 结构数组的定义

在 C51 语言中，结构数组的声明方法和结构变量相类似，示例如下：

```
struct student                                //定义结构
{
char *name;                                  //姓名
int num;                                     //学号
int age;                                     //年龄
char *sex;                                   //性别
float score;                                //分数
};
struct student boy[5];                       //声明结构数组
```

也可以在定义结构的同时声明结构数组。示例如下：

```
struct student                                //定义结构
{
char *name;                                  //姓名
int num;                                     //学号
int age;                                     //年龄
char *sex;                                   //性别
float score;                                //分数
}boy[5];                                     //声明结构数组
```

以上两种方式是等价的。这里定义的结构数组 boy，共有 5 个元素，boy[0]~boy[4]，每个数组元素都具有 struct student 的结构形式。

9.6.2 结构数组的初始化赋值

同结构变量一样，在声明结构数组的同时，可以对其进行初始化赋值。示例如下：

```
struct student                                //定义结构
{
char *name;                                  //姓名
int num;                                     //学号
int age;                                     //年龄
char *sex;                                   //性别
float score;                                //分数
}boy[5]={                                     //声明并初始化结构数组
{"Alice",100,24,"F",90.5},
{"Bob",101,24," M ",90.0},
{"Jack",102,25," M ",88.5},
{"Twis",103,23," M ",78.5},
{"Bohm",104,21," M ",82.5}};
```

下面举一个具体的例子，介绍其用法。示例如下：

```
#include <stdio.h>                            //头文件

struct student                                //定义结构
{
```

51 单片机开发与应用技术详解

```
char *name;                //姓名
int num;                   //学号
int age;                   //年龄
char *sex;                 //性别
float score;               //分数
}boy[5]={                  //声明并初始化结构数组
{"Alice",100,24,"F",90.5},
{"Bob",101,24," M ",90.0},
{"Jack",102,25," M ",88.5},
{"Twis",103,23," M ",78.5},
{"Bohm",104,21," M ",82.5}};

void main()                //主函数
{
int i;
for(i=0;i<5;i++)          //循环
{
    if(boy[i].score>=90.0)  //判断分数是否大于 90，如果大于则输出
        printf("%s %d %d %s %f \n",boy[i].name,boy[i].num,boy[i].age,boy[i].sex,
boy[i].score);
}
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Alice 100 24 F 90.500000
Bob 101 24 M 90.000000
```

该程序用于统计分数大于 90.0 的学生。在本程序中，一开始便声明并初始化了结构变量数组 boy[5]。然后，在主程序中，通过一个循环判断语句，判断每个学生的分数是否大于 90.0，如果大于则输出该学生的全部信息。

9.7 结构指针

结构指针是指该指针变量用来指向一个结构变量。其中，结构指针变量中的值是所指向的结构变量的首地址。在 C51 语言中，通过结构指针可以访问该结构变量，这与数组指针是类似的。结构指针变量声明的一般形式如下：

```
struct 结构名 *结构指针变量名
```

其中，struct 为关键字，结构名为结构指针的类型。例如，对于前面的实例中定义的 student 结构，如果要声明一个指向 student 的指针变量 pstu，可采用如下的方式。

```
struct student *pstu;
```

当然也可在定义 student 结构时同时声明结构指针 pstu，示例如下：

```
struct student            //定义结构
{
char *name;               //姓名
int num;                  //学号
int age;                  //年龄
char *sex;                //性别
float score;              //分数
}*pstu;                   //声明结构指针
```

结构指针变量必须要先赋值后才能程序中使用。赋值操作是把结构变量的首地址赋予该

结构指针变量。

结构名和结构变量是两个不同的概念，不能混淆。结构名只能表示一个结构形式，编译系统并不对它分配内存空间。结构变量是被声明为某种类型的结构的变量，编译程序可以对该变量分配存储空间。

这里需要注意，程序中不能把结构名赋予该指针变量。例如，Alice 是被说明为 student 类型的结构变量，则 `pstu=&Alice` 是正确的，而 `pstu=&student` 是错误的。因为不可能去取一个结构名的首地址。

有了结构指针变量，就能更方便地访问结构变量的各个成员。其访问的一般形式如下：

`(*结构指针变量). 成员名`

或者如下：

`结构指针变量->成员名`

其中，“.”和“->”为成员符，成员名即结构中对应的成员。应该注意第一种方式中的括号不可少，因为成员符“.”的优先级高于“*”。如去掉括号写作“*结构指针变量. 成员名”则等效于“*(结构指针变量. 成员名)”，这样意义就完全不对了。

对于前面声明的结构指针变量 `pstu`，可以采用如下的两种方式来访问 `num` 成员，示例如下：

```
(*pstu).num
pstu->num
```

下面通过例子来说明结构指针变量的具体声明和使用方法。程序示例如下：

```
#include <stdio.h>                                //头文件
struct student                                    //定义结构
{
    char *name;                                    //姓名
    int num;                                       //学号
    int age;                                       //年龄
    char *sex;                                    //性别
    float score;                                  //分数
}Bob={"Bob",101,24," M ",90.0};                  //声明并初始化结构变量

void main()                                       //主函数
{
    struct student *pstu;                         //定义结构指针
    pstu=&Bob;                                    //为结构指针赋值
                                                //采用方式 1 输出成员值
    printf("Name=%s,Num=%d\n",Bob.name,Bob.num);
    printf("Age=%d,Sex=%s,score=%f\n",Bob.age,Bob.sex,Bob.score);
                                                //采用方式 2 输出成员值
    printf("Name=%s,Num=%d\n",(*pstu).name,(*pstu).num);
    printf("Age=%d,Sex=%s,score=%f\n",(*pstu).age,(*pstu).sex,(*pstu).score);
                                                //采用方式 3 输出成员值
    printf("Name=%s,Num=%d\n",pstu->name,pstu->num);
    printf("Age=%d,Sex=%s,score=%f\n",pstu->age,pstu->sex,pstu->score);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Name=Bob,Num=101
Age=24,Sex= M ,score=90.000000
Name=Bob,Num=101
Age=24,Sex= M ,score=90.000000
```

51 单片机开发与应用技术详解

```
Name=Bob,Num=101
Age=24,Sex= M ,score=90.000000
```

在本程序中，首先定义了结构 **student**，并同时声明并初始化了结构变量 **Bob**。然后，在主程序中定义结构指针并赋值。最后，通过三种不同的方式输出结构变量中的成员值。

从本例的运行结果可以看出，如下的三种用于表示结构成员的形式是完全等效的。

- 结构变量.成员名
- (*结构指针变量).成员名
- 结构指针变量->成员名

结构指针变量可以指向一个结构数组，这时结构指针变量的值是整个结构数组的首地址。结构指针变量也可指向结构数组的一个元素，这时结构指针变量的值是该结构数组元素的首地址。设 **ps** 为指向结构数组的指针变量，则 **ps** 也指向该结构数组的 0 号元素，**ps+1** 指向 1 号元素，**ps+i** 则指向 **i** 号元素。这与普通数组的情况是一致的。例如，用指针变量输出结构数组，程序示例如下：

```
#include <stdio.h>                                //头文件
struct student                                    //定义结构
{
    char *name;                                    //姓名
    int num;                                       //学号
    int age;                                       //年龄
    char *sex;                                    //性别
    float score;                                  //分数
}boy[4]={                                          //初始化
    {"Bob",101,24,"M",90.0},
    {"Jack",102,25,"M",88.5},
    {"Twis",103,23,"M",78.5},
    {"Bohm",104,21,"M",82.5}};

void main()                                       //主函数
{
    struct student *pstu;                        //定义结构指针
    printf("Name\tNum\tAge\tSex\tscore\n");      //输出
    for(pstu=boy;pstu<boy+4;pstu++)             //输出各元素的成员值
        printf("%s\t%d\t%d\t%s\t%f\n",(*pstu).name,(*pstu).num,pstu->age,pstu->sex,ps
tu->score);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

Name	Num	Age	Sex	score
Bob	101	24	M	90.000000
Jack	102	25	M	88.500000
Twis	103	23	M	78.500000
Bohm	104	21	M	82.500000

本程序一开始定义了结构 **student**，同时声明并初始化了结构数组 **boy[4]**。程序中分别采用了两种方式输出各个元素的成员值。

注意一个结构指针变量虽然可以用来访问结构变量或结构数组元素的成员，但是不能使它指向一个成员，也就是说不允许取一个成员的地址来赋予。示例如下：

```
ps=&boy[1].sex;                                //是错误的。
ps=boy;                                         //是正确的，赋予数组首地址
ps=&boy[0];                                    //是正确的，赋予 0 号元素首地址
```

9.8 特殊结构

在 C51 语言中，除了前面介绍的基本的结构外，还有嵌套结构和位结构两种特殊形式的结构。

9.8.1 嵌套结构

嵌套结构是指在一个结构成员中可以包括其他一个结构。这一点在前面曾经提到，这里进行详细的说明。嵌套结构的定义示例如下：

```
struct kemu                                //定义结构 kemu
{
    float Chinese;                        //中文成绩
    float English;                       //英语成绩
    float Math;                          //数学成绩
};
struct student                            //定义结构 student
{
    char *name;                          //姓名
    int num;                             //学号
    int age;                             //年龄
    char *sex;                           //性别
    struct kemu score;                   //成绩
};
```

在本例中，首先定义了一个结构 `kemu`，由 `Chinese`（中文）、`English`（英文）和 `Math`（数学）三个成员组成，表示课程信息。然后定义了结构 `student`，其中的最后一个成员 `score` 是一个 `kemu` 结构成员，即实现了嵌套结构。嵌套式结构成员的表达方式如下：

结构变量名.嵌套结构变量名.结构成员名

其中嵌套结构可以有很多，结构成员名为最内层结构中的成员名。嵌套结构的成员名的引用，是从最外层直到最内层逐个被列出的。

9.8.2 位结构

位结构是一种特殊的结构，用于访问一个字节或字的多个位。位结构定义的一般形式如下：

```
struct 位结构名{
    数据类型 变量名: 整型常数;
    数据类型 变量名: 整型常数;
}位结构变量;
```

其中，数据类型必须是 `int`（`unsigned` 或 `signed`）；整型常数必须是非负的整数，范围是 0~15，表示二进制位的个数，即表示有多少位；变量名是可选项，可以不命名，这样规定是为了排列需要。例如，下面定义了一个位结构。

```
struct{
    unsigned LBitStu: 8;                 //LBitStu 占用低字节的 0~7 共 8 位
    unsigned HBitStu 1: 4;              //HBitStu 1 占用高字节的 0~3 位共 4 位
    unsigned HBitStu 2: 3;              //HBitStu 2 占用高字节的 4~6 位共 3 位
    unsigned HBitStu 3: 1;              //HBitStu 3 占用高字节的第 7 位
}BitStu;
```

51 单片机开发与应用技术详解

位结构成员的访问与结构成员的访问相同。例如，访问上例位结构中的 `LBitStu` 成员可写成如下形式。

```
BitStu.LBitStu
```

在 C51 语言程序中需要按位访问数据的位时，采用位结构比使用位运算符更加方便。在使用位结构时，应注意以下几点。

- 位结构中的成员既可以定义为 `unsigned`，也可定义为 `signed`。
- 位结构总长度（位数）是各个位成员定义的位数之和，可以超过两个字节。
- 但当成员长度为 1 时，会被认为是 `unsigned` 类型，因为单个位不可能具有符号。
- 位结构中的成员不能使用数组和指针，但位结构变量可以是数组和指针。如果是指针，其成员访问方式同结构指针。
- 位结构成员可以与其他结构成员一起使用。示例如下：

```
struct student                                //定义结构
{
char *name;                                  //姓名
int num;                                     //学号
int age;                                     //年龄
char *sex;                                   //性别
float score;                                //分数
unsigned zhuce: 1;                           //注册
unsigned pay: 1;                             //交费
};
```

本例中的结构定义了关于一个学生的信息。其中有两个位结构成员，第一个成员 `zhuce` 表示该学生是否已经注册，第二个成员 `pay` 表示该学生是否已经缴纳学费。每个位结构成员只有一位，因此只占一个字节但保存了两个信息，由此可见，合理使用位结构可以节省存储空间。

9.9 小结

本章首先介绍了 C51 语言中的结构的使用，主要包括结构的定义、结构变量的定义和赋值。接着，本章详细讲解了结构数组及结构指针的使用。最后，本章还对几种特殊的指针进行了详细的介绍，主要包括嵌套结构和位结构。结构是 C51 语言中重要的数据类型，其和 C 语言中的结构非常类似。熟练掌握本章内容是学习 C51 语言的基础，同时对读者以后的 C51 程序设计工作会大有帮助。

第 10 章 联合、枚举、类型说明和位域

在 C51 语言中，除了数组和指针外，还定义了联合和枚举两种聚合数据类型，并且可以进行类型说明和定义位域，这些是特殊形式的数据类型。下面分别进行讲解。

10.1 联合类型

联合类型，是指将不同的变量组织成一个整体的数据类型。其中的这些变量在内存中占用同一段存储单元，而在不同的时间保存不同的数据类型和不同长度的变量。因此，联合类型也称为共用体。

10.1.1 联合和联合变量的定义

联合的定义与结构十分相似，其形式如下：

```
union 联合名
{
    数据类型 成员名;
    数据类型 成员名;
    ...
};
```

其中，`union` 是联合类型的关键字。同结构一样，可以在定义联合的同时声明一个联合变量，示例如下：

```
union 联合名
{
    数据类型 成员名;
    数据类型 成员名;
    ...
} 联合变量名;
```

另外，也可以单独声明联合变量，示例如下：

```
union char_int
{
    char ch;
    int i;
};
union char_int a;
```

其中，联合变量 `a` 中，字符型成员 `ch` 和整型成员 `i` 所占的内存大小一致，共用同一个内存位置。如果联合变量中成员变量的大小不一致，则程序在编译的时候，自动取其中最大的长度

为整个联合变量的长度。示例如下：

```
union ThreeInOne
{
    int a ;
    char b;
    float c;
}Test;
```

本例定义了一个名为 `ThreeInOne` 的联合，并定义了一个名为 `Test` 联合变量。其中最长的成员是浮点型的 `c`，因此联合变量的长度与 `c` 一致，即占用 4 个字节。

10.1.2 联合变量成员的引用

联合体变量成员的引用方法与结构体完全相同。例如，对于前面定义的 `Test`，其成员引用为 `Test.a`、`Test.b`、`Test.c`。示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    union ThreeInOne                               //定义联合并声明联合变量
    {
        int a ;
        char b;
        float c;
    }Test;
    Test.a=8;                                       //使用成员变量 a
    printf("Test.a=%d\n",Test.a);
    //printf("Test.b=%f\n",Test.c);                //无意义
    Test.c=6.7;                                     //使用成员变量 b
    printf("Test.c=%f\n",Test.c);
    Test.b='A';                                     //使用成员变量 c
    printf("Test.b=%c\n",Test.b);
    printf("Test.a=%d\n",Test.a);                  //无意义
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Test.a=8
Test.c=0.000000
Test.c=6.700000
Test.b=A
```

在本程序中，首先为成员变量 `a` 赋值，此时引用 `a` 是有效的，第二个输出语句引用了成员变量 `b` 是无效的，因此输出零结果。

❗注意：不能同时引用三个成员，既在任意时刻，只能使用其中之一的成员。

在 C51 语言中，联合变量也可以定义成数组或指针。当联合变量定义为指针时，要用 “->” 符号访问成员，可表示成如下的形式。

联合名->成员名

下面举一个使用联合变量数组的程序，示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
```

```

union ThreeInOne                                //定义联合
{
int a;
char b;
float c;
};

union ThreeInOne un[3];                          //声明联合数组变量

un[0].a=10;                                       //赋值
un[1].b=3.7;
un[2].c='A';

printf("un[0].a=%d\n",un[0].a);                  //输出成员变量
printf("un[1].b=%c\n",un[1].b);
printf("un[2].c=%f\n",un[2].c);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

un[0].a=10
un[1].b=A
un[2].c=3.700000

```

在本程序中，首先声明一个联合变量数组 `un`，然后对其中的每个元素进行赋值，最后分别打印输出结果。这里再举一个使用联合变量指针的程序，示例如下：

```

#include <stdio.h>                                //头文件

void main()                                       //主函数
{
int i=10;                                       //定义并声明变量
float f=3.7;
char c='A';

union ThreeInOne                                //定义联合
{
int a;
char b;
float c;
};

union ThreeInOne *un;                           //定义联合指针
un=&i;                                           //赋值

printf("un->a=%d\n",un->a);                      //输出
un=&c;
printf("un->b=%c\n",un->b);
un=&f;
printf("un->c=%f\n",un->c);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

un->a=10
un->b= A
un->c=3.700000

```

在本程序中，首先声明一个联合变量指针 `un`，然后分别对 `un` 进行赋值，并打印输出结果。

在 C51 语言中，联合和结构可以嵌套使用，即联合可以出现在结构内，联合内的成员也可以是结构。示例如下：

```

union ThreeInOne
{
int a;
char b;
float c;
struct

```

```
{
    int a;
    char b;
    float c;
}st;
}un;
```

这个例子是结构出现在联合中，成员变量的引用为 `un.a`、`un.b`、`un.c`、`un.st.a`、`un.st.b` 和 `un.st.c`。在下面的这个例子中，联合出现在结构中，示例如下：

```
struct ThreeInOne
{
    int a;
    char b;
    float c;
    union
    {
        int a;
        char b;
        float c;
    }un;
}st;
```

对于上面的结构，成员变量的引用为 `st.a`、`st.b`、`st.c`、`st.un.a`、`st.un.b` 和 `st.un.c`。

10.2 结构和联合的区别

结构和联合在很多方面都很相似，但它们之间有本质的区别。结构和联合的区别主要体现在如下几点。

- 对于由多个不同数据类型成员组成的结构变量和联合变量，在任何同一时刻，结构的所有成员都存在，而联合变量中只存放了一个被选中的成员。
- 结构变量的不同成员赋值是互不影响的，而对于联合变量的不同成员赋值，将会对其其他成员重写，原来成员的值就不存在了。

从数据存储角度来说，结构和联合变量的区别主要在于联合变量的成员占用同一个内存空间，而结构变量中的成员分别独占自己的内存空间，互相不干扰，如图 10-1 所示。下面举一个例子来加深对联合的理解，程序示例如下：

```
#include <stdio.h>                                //头文件
union                                              //定义联合
{
    int n;
    struct
    {
        char first;
        char second;
    }st;
}num;                                              //声明结构变量
void main()                                       //声明联合变量 num
{
    num.n=0x6261;                                //主函数
    printf("%c %c\n",num.st.first,num.st.second); //赋值
    num.st.first='A';                             //输出
    num.st.second='B';                             //赋值
    printf("0x%x\n",num.n);                       //输出
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
b a
0x4142
```

在本例中，当给成员 `n` 赋值后，其低字节和高字节也就是 `first` 和 `second` 的值；当给 `first` 和 `second` 赋字符后，这两个字符的 ASCII 码也将作为 `n` 的低字节和高字节。

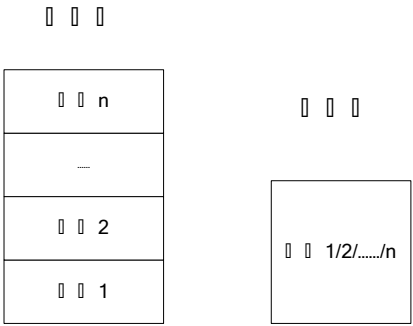


图 10-1 结构类型和联合类型所占的存储空间比较

10.3 枚举类型

枚举类型主要用于将变量的取值限定在一个有限的范围内的场合。枚举类型在定义中列举出所有可能的取值，被声明为该类型的变量取值不能超过定义的范围。枚举类型是一种基本数据类型，而不是一种构造类型。例如，一个星期内只有七天、一年只有十二个月、一个班每周有六门课程等。如果把这些量说明为整型、字符型或其他类型显然是不妥当的。此时，采用枚举类型不但可以简洁明了地表示其取值，而且节省存储空间。

10.3.1 枚举的定义

在 C51 语言中，枚举类型定义的一般形式如下：

```
enum 枚举名
{
    标识符 [=整型常量] ,
    标识符 [=整型常量] ,
    ...
} 枚举变量;
```

其中，`enum` 为关键字，枚举名即所定义的枚举类型，标识符是枚举类型的取值范围，枚举变量是声明为该枚举类型的变量。每个枚举元素后的结束符是“`,`”，而不是“`;`”，并且最后一个元素后的结束符可以省略不写。以一个星期内的七天为例，枚举类型的定义示例如下：

```
enum weekday //定义枚举类型
{
    Monday, //枚举元素, 0
    Tuesday, //枚举元素, 1
    Wednesday, //枚举元素, 2
    Thursday, //枚举元素, 3
    Friday, //枚举元素, 4
    Saturday, //枚举元素, 5
```

```
Sunday,                                //枚举元素, 6
};
```

其中, 枚举名为 `weekday`, 枚举元素共有 7 个, 对应一周中的七天。在 C51 程序中, 凡被声明为 `weekday` 类型的变量取值只能是七天中的某一天。

10.3.2 枚举变量的声明

在 C51 语言中, 枚举变量的声明有三种方式。这里假设有变量 `a`、`b`、`c` 被声明为上面定义的 `weekday` 类型, 分别采用下面三种声明方式。

➤ 先定义后声明, 示例如下:

```
enum weekday                            //定义枚举类型
{
Monday,                                //枚举元素, 0
Tuesday,                              //枚举元素, 1
Wednesday,                            //枚举元素, 2
Thursday,                             //枚举元素, 3
Friday,                               //枚举元素, 4
Saturday,                             //枚举元素, 5
Sunday,                               //枚举元素, 6
};
enum weekday a,b,c;                    //声明枚举变量
```

➤ 在定义时声明, 示例如下:

```
enum weekday                            //定义枚举类型
{
Monday,                                //枚举元素, 0
Tuesday,                              //枚举元素, 1
Wednesday,                            //枚举元素, 2
Thursday,                             //枚举元素, 3
Friday,                               //枚举元素, 4
Saturday,                             //枚举元素, 5
Sunday,                               //枚举元素, 6
}a,b,c;                                //声明枚举变量
```

➤ 直接声明, 示例如下:

```
enum
{
Monday,                                //枚举元素, 0
Tuesday,                              //枚举元素, 1
Wednesday,                            //枚举元素, 2
Thursday,                             //枚举元素, 3
Friday,                               //枚举元素, 4
Saturday,                             //枚举元素, 5
Sunday,                               //枚举元素, 6
}a,b,c;                                //声明枚举变量
```

10.3.3 枚举类型变量的赋值

枚举类型在定义时如果没有初始化，即省去“=整型常数”，枚举元素本身由系统定义了一个表示序号的数值，从 0 开始顺序定义为 0, 1, 2…。如果枚举中的某元素被赋值后，其后成员按依次加 1 的规则，确定其值。例如，在 weekday 中，依次 Monday 值为 0, Tuesday 值为 1, …, Sunday 值为 6。示例如下：

```
#include <stdio.h>                                //头文件

void main()                                       //主函数
{
    enum weekday                                //定义枚举类型
    {
        Monday,                                //枚举元素, 0
        Tuesday,                               //枚举元素, 1
        Wednesday,                             //枚举元素, 2
        Thursday,                              //枚举元素, 3
        Friday,                                //枚举元素, 4
        Saturday,                              //枚举元素, 5
        Sunday,                                //枚举元素, 6
    }a,b,c;                                     //声明枚举变量

    a=Monday;                                   //变量赋值
    b=Friday;
    c=Sunday;

    printf("a=%d,b=%d,c=%d\n",a,b,c);          //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
a=0,b=4,c=6
```

程序中，首先定义枚举类型 weekday，同时声明了枚举变量 a、b 和 c。接着，对枚举变量分别赋值，并打印输出其对应的值。

枚举类型在使用中要注意以下几点。

- 枚举值是常量，不是变量。不能在程序中用赋值语句再对它赋值。例如，对枚举 weekday 的元素再作以下赋值，这些都是错误的。

```
Sunday=5;Monday=2;Sunday=Monday;
```

- 只能把枚举元素赋予枚举变量，不能把枚举元素的数值直接赋予枚举变量。示例如下：

```
a=Sunday;b=Monday;                                //是正确的
a=0;b=1;                                           //是错误的
```

如果一定要把枚举元素的数值赋予枚举变量，则必须用强制类型转换。示例如下：

```
a=(enum weekday)2;
```

相当于

```
a=Wednesday;
```

- 枚举元素不是字符常量也不是字符串常量，使用时不要加单、双引号。
- 初始化赋值时，可以为其赋负数，其后成员仍然按依次加 1 的规则确定其值。示例如下：

```
#include <stdio.h>                                //头文件
```

```
void main()                                //主函数
{
enum                                        //定义枚举并声明枚举变量
{
t1,
t2=-2,                                    //枚举元素赋初值
t3,
t4=10,
t5,
}t;

t=t1;                                     //枚举变量赋值

printf("t1=%d\n",t);                      //输出结果
t=t2;
printf("t2=%d\n",t);
t=t3;
printf("t3=%d\n",t);
t=t4;
printf("t4=%d\n",t);
t=t5;
printf("t5=%d\n",t);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
t1=0
t2=-2
t3=-1
t4=10
t5=11
```

上面程序中，首先声明了枚举变量 t，其中对枚举元素 t2 赋了负的初值。最后，分别对 t 进行赋值并打印输出其定义的枚举元素值。

10.4 类型说明

类型说明是指由用户自己定义类型说明符，也就是说允许由用户为数据类型取新类型名。类型说明的一般形式如下：

```
typedef 原类型名 新类型名；
```

其中 typedef 为类型定义符，原类型名中含有定义部分，新类型名一般用大写表示，以便于区别。例如，有整型变量 a、b，其声明如下：

```
int a,b;
```

其中 int 是整型变量的类型说明符。int 的完整写法为 integer。为了增加程序的可读性，可把整型声明符用 typedef 重新定义如下：

```
typedef int INTEGER;
```

这样，在程序中就可用 INTEGER 来代替 int 作整型变量的类型声明了。示例如下：

```
INTEGER a,b;                                //等效于：int a,b;
```

用 typedef 定义数组、指针、结构等类型将带来很大的方便。示例如下：

```
typedef char NAME[20];
```

这里用 NAME 来表示是字符数组类型，数组长度为 20。然后可用 NAME 声明变量，示例如下：

```
NAME a1,a2,s1,s2;
```

该语句完全等效于下面的语句。

```
char a1[20],a2[20],s1[20],s2[20]
```


可以看出，使用类型说明可以使程序书写简单，意义更为明确，可读性得到增强。又如下面的语句。

```
typedef struct stu
{
char name[20];
int age;
char sex;
} STU;
```

这里定义 STU 表示 stu 的结构类型，然后便可以用 STU 来说明结构变量。

```
STU Alice,Bob;
```

使用类型说明的另一个优点是，便于程序的移植。

 **说明：**有时也可用宏定义来代替 typedef 的功能，但是宏定义是由预处理完成的，而 typedef 则是在编译时完成的，后者更为灵活方便。

10.5 位域

“位域”是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位来表示。因为有些信息在存储时，只需占几个或一个二进制位，而并不需要占用一个完整的字节。例如在存放一个开关量时，只有 0 和 1 两种状态，用一位二进制位即可。因此，采用位域可以节省存储空间，并使处理简便。位域数据结构，也常称为“位段”。

10.5.1 位域的定义和位域变量的声明

位域定义与结构定义相似，其一般形式如下：

```
struct 位域结构名
{
类型说明符 位域名：位域长度；
...
};
```

其中，struct 为关键字，在位域结构中说明了位域名的类型及位域的长度。示例如下：

```
struct btest
{
int a:4;
int b:1;
int c:3;
};
```

其中，定义了位域类型 btest，包含三个位段 a、b 和 c，长度分别为 4、1 和 3。位域变量的声明与结构变量说明的方式相同，可以采用如下三种方式。

➤ 定义的同时进行声明，示例如下：

```
struct btest
{
int a:4;
int b:1;
int c:3;
}data;
```

➤ 先定义后声明，示例如下：

```
struct btest
```



```
{
int a:4;
int b:1;
int c:3;
};
btest data;
```

➤ 直接进行声明，示例如下：

```
struct
{
int a:4;
int b:1;
int c:3;
}data;
```

上面的例子中定义变量 **data** 为 **btest** 位域变量，共占一个字节。其中位域 **a** 占 4 位，位域 **b** 占 1 位，位域 **c** 占 3 位。在定义位域的时候，需要注意如下几点。

➤ 位域定义时可以省略位域名，这时它只用来填充或调整位置。无名的位域是不能使用的。示例如下：

```
struct btest
{
int a:1;
int :2 ; //该 2 位不能使用，为空域
int b:3;
int c:2;
};
```

➤ 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。示例如下：

```
struct btest
{
unsigned a:4;
unsigned :0; //空域
unsigned b:5; //从下一单元开始存放
unsigned c:3;
}
```

在这个位域定义中，**a** 占第一字节的 4 位，后 4 位填 0 表示不使用，**b** 从第二字节开始，占用 5 位，**c** 占用 3 位。

➤ 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进制位。

从以上分析可以看出，位域在本质上就是一种结构类型，不过其成员是按二进制位进行分配的。位域特别适合于需要位操作的程序中。

10.5.2 位域变量的使用

位域变量的位域名引用的一般形式如下：

位域变量名.位域名

位域允许用各种格式输出，也可以使用指针。位域变量在程序中的使用，示例如下：

```
#include <stdio.h> //头文件

void main() //主函数
{
```

```

struct btest                                //定义位域
{
    unsigned a:1;
    unsigned b:3;
    unsigned c:4;
}mybtest,*pbit;                             //声明位域变量和位域指针

mybtest.a=1;                                //位域变量赋值
mybtest.b=7;
mybtest.c=15;

printf("%d,%d,%d\n",mybtest.a,mybtest.b,mybtest.c); //输出结果

pbit=&mybest;                               //位域指针赋值

pbit->a=0;                                   //执行运算
pbit->b&=3;
pbit->c|=1;

printf("%d,%d,%d\n",pbit->a,pbit->b,pbit->c);      //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

1,7,15
0,3,15

```

本例中，定义了位域结构 `btest`，三个位域为 `a`、`b`、`c`，同时声明了 `btest` 类型的变量 `mybtest` 和指向 `btest` 类型的指针变量 `pbit`。程序中，分别采取了两种方式进行位域变量的操作。首先给位域变量的每个成员赋值，然后输出其值的大小。然后将 `mybtest` 的地址赋给指针变量 `pbit`，接着分别采用复合运算符“`&=`”和“`|=`”对成员变量进行位运算，最后使用指针输出结果。

10.6 小结

本章主要介绍了几种特殊形式的聚合数据类型，包括联合类型、枚举类型，还介绍了类型说明及位域。这些特殊的数据类型是对基本数据类型的有效扩充。灵活运用这些特殊数据类型，可以方便单片机的程序设计。因此，熟练掌握本章内容是单片机程序设计的基础。

第 11 章 C51 语言的函数

C51 语言中也引入了函数的概念。C51 语言程序中的函数数目是不受限制的，但是，一个典型的单片机程序都必须包含且只能包含一个主函数，即 `main` 函数，它是整个程序的入口，整个程序从这个主函数开始执行。

C51 语言允许用户使用一些编译环境自带的库函数，合理使用库函数可以大大简化程序设计过程。同时，为了有利于程序的模块化，促进程序资源的共享。C51 语言也支持用户使用自定义函数。本章将详细介绍函数的概念、调用、作用域等，并对 `main` 函数和 C51 语言常用的库函数进行详细的讲解。

11.1 函数的概念和分类

函数是能够实现特定功能的代码段。一个 C51 语言程序通常由一个主函数和若干个函数构成。其中，主函数即 `main` 函数。C51 语言程序的执行总是从 `main` 函数开始，完成对其他函数的调用后再返回到主函数，最后由 `main` 函数结束整个程序。一个 C51 语言源程序必须且只能有一个主函数 `main`。

除了主函数外，C51 语言还提供了极为丰富的库函数，而且还允许用户自定义函数。在 C51 程序中，由主函数调用其他函数，其他函数之间也可以相互调用。同一个函数可以被一个或多个函数调用任意次。

在使用 C51 语言函数时，需要注意如下几点。

- C51 语言的源程序的函数数目是不限的。
- 在一个函数的函数体内，不能再定义另一个函数，即不能嵌套定义。
- 函数之间允许相互调用，也允许“嵌套调用”。
- 函数还可以自己调用自己，称为“递归调用”。
- `main` 函数是主函数，它可以调用其他函数，而不允许被其他函数调用。

在 C51 语言中，可从不同的角度对函数分类，下面分别进行介绍。

11.1.1 从函数定义角度

从函数定义的角度看，函数可分为库函数和用户自定义函数两种。分别介绍如下：

对于库函数，其是由 C51 语言系统提供，用户无须定义，只需在主程序前包含有该函数原型的头文件即可在程序中直接调用。这些库函数又可从功能角度分为如下几类。

- 字符类型分类函数：用于对字符按 ASCII 码分类为字母、数字、控制字符、分隔符、大小写字母等。
- 转换函数：用于字符或字符串的转换，在字符量和各类数字量（整型、实型等）之间进行转换，在大、小写之间进行转换。
- 输入输出函数：用于完成输入输出功能，如 `printf`、`scanf` 等。
- 字符串函数：用于字符串操作和处理。
- 数学函数：用于数学函数计算。
- 其他函数：用于其他各种功能。

对于用户自定义函数，需要用户自行编写。在使用自定义函数时，不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

11.1.2 从有无返回值角度

从有无返回值角度来划分，又可把 C51 语言函数分为有返回值函数和无返回值函数两种。分别介绍如下：

- 有返回值函数：此类函数被调用执行完后，将向调用者返回一个执行结果，称为“函数返回值”。库函数中包含多个带有返回值的函数。另外，由用户定义的这种要返回函数值的函数，必须在函数定义和函数说明中明确返回值的类型。
- 无返回值函数：相当于其他高级语言中的过程。此类函数用于完成某项特定的任务，执行完成后不向调用者返回函数值。库函数中包含多个不带有返回值的函数。而对于用户自定义的无返回值函数，可指定它的返回为“无值型”，其类型说明符为“void”。

11.1.3 从数据传送角度

从主调函数和被调函数之间数据传送的角度来划分，又可把 C51 语言函数分为无参函数和有参函数两种。分别介绍如下：

- 无参函数：主调函数和被调函数之间不进行参数传送，因此在函数定义、函数说明及函数调用中也就可以不带参数。此类函数通常用来完成一组指定的功能，可以带有返回值，也可以没有返回函数值。
- 有参函数：主调函数和被调函数之间存在参数传送，因此在函数定义及函数说明时都需要有参数，称为“形式参数”（简称为“形参”）。在主调函数中进行函数调用时也必须给出参数，称为“实际参数”（简称为“实参”）。在函数调用时，主调函数将把实参的值传送给形参，供被调函数使用。有参函数可以带有返回值，也可以没有返回函数值。

11.2 函数的定义

在 C51 语言中，函数定义的一般形式如下：

类型说明符 函数名 (形式参数列表)

形参类型说明

```
{
    语句
    return 语句
}
```

其中：

- “类型说明符”声明了函数返回值的类型，该返回值可以是任何有效类型。如果没有类型说明符出现，函数返回一个整型值。如果函数没有返回值，则可以采用 void 说明符。函数类型的说明必须处于对它的首次调用之前，这样 C51 语言程序编译时才能为返回非整型值的函数生成正确代码。
- “形式参数列表”是一个用逗号分隔的参数变量表。当函数被调用时，这些变量接收调用参数的值。如果函数是无参函数，这时函数表是空的，但括号仍然是必须要有的。
- “形参类型说明”声明了函数内部参数的类型。其数据类型可以为 C51 语言支持的数据类型。
- “return 语句”用于返回函数执行的结果，如果没有返回值，则可以省略该语句。

函数与变量一样，在使用前必须先定义。这里举例说明自定义函数在程序中的应用，示例如下：

```
#include <stdio.h>
```

```
//头文件
```

51 单片机开发与应用技术详解

```
int max(a,b)                                //定义函数，用于求两个数中的最大值
int a,b;                                    //形参类型说明
{
    if(a>b)                                  //if 判断语句
        return a;                            //如果 a 大于 b，则函数返回 a
    else
        return b;                            //如果 a 小于 b，则函数返回 b
}

void main()                                  //主函数
{
    int a=7,b=5;                             //定义并初始化变量
    printf("a=%d,b=%d",a,b);                 //输出 a、b 的值
    printf("max(a,b)=%d\n",max(a,b));         //调用函数并输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
a=7,b=5
max(a,b)=7
```

在本程序中，首先定义 max 函数是一个整型函数，形参为 a 和 b，其函数返回值是一个整数。形参 a 和 b 的类型均为整型量。在{}中的函数体内是 max 函数语句部分，其使用 return 语句把最大值作为函数的返回值返回给主调函数。在主调函数 main 中调用 max 函数，并输出比较结果。

另外，C51 语言中的函数定义还可以采用如下的方式。

类型说明符 函数名 (形参类型说明，形式参数列表)

```
{
    语句
    return 语句
}
```

其中，将形参类型说明放在参数列表中一并进行说明。这样可以增强函数的可读性，避免一些参数变量声明上的错误。这里采用上面的例子，举例说明这种函数定义方式在程序设计中的应用。程序示例如下：

```
#include <stdio.h>                            //头文件
int max(int a,int b)                          //定义函数，用于求两个数中的最大值
{
    if(a>b)                                    //if 判断语句
        return a;                            //如果 a 大于 b，则函数返回 a
    else
        return b;                            //如果 a 小于 b，则函数返回 b
}

void main()                                  //主函数
{
    int a=7,b=5;                             //定义并初始化变量
    printf("a=%d,b=%d",a,b);                 //输出 a、b 的值
    printf("max(a,b)=%d\n",max(a,b));         //调用函数并输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
a=7,b=5
max(a,b)=7
```

在 C51 程序中，一个函数的定义可以放在任意位置，既可放在 main 主函数之前，也可放在主函数之后。如果放在主函数之后，需要在主函数执行之前对该函数进行声明。示例如下：

```
#include <stdio.h>                                //头文件
int max(int a,int b)                               //声明函数

void main()                                        //主函数
{
    int a=7,b=5;                                   //定义并初始化变量
    printf("a=%d,b=%d",a,b);                      //输出 a、b 的值
    printf("max(a,b)=%d\n",max(a,b));              //调用函数并输出结果
}

int max(int a,int b)                               //定义函数，用于求两个数中的最大值
{
    if(a>b)                                         //if 判断语句
        return a;                                 //如果 a 大于 b，则函数返回 a
    else
        return b;                                 //如果 a 小于 b，则函数返回 b
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
a=7,b=5
max(a,b)=7
```

该程序中，在主函数之前便对自定义的 max 函数进行了声明，而 max 函数则放在主函数之后进行定义。程序具有同样的执行效果。

！注意：对库函数的调用不需要再做说明，但必须把该函数的头文件用 include 命令包含在源文件前部。

11.3 函数的参数

函数的参数用于主调函数与被调函数间进行的数据交换。在 C51 语言中，实参和形参都可以采用多种数据类型。函数的参数是个很重要的内容。

11.3.1 形参和实参

在函数调用过程中，函数的参数可以分为形参和实参两种。发生函数调用时，主调函数把实参的值传送给被调函数的形参，从而实现主调函数向被调函数的数据传送。函数的形参和实参在使用时需要注意如下几点。

- 实参和形参在数量、类型、顺序上应保持严格一致，否则会因类型不匹配而导致错误。
- 形参只有在函数内部有效。因为形参变量只有在被调用时才分配内存单元，在调用结束后，将立即释放内存单元。因此，函数调用结束并返回主调函数后，则不能再使用该形参变量。
- 实参可以是常量、变量、表达式等，无论实参是何种类型的量，在进行函数调用时，它们都必须先赋予确定的值，以便把这些值传送给形参。实参出现在主调函数中，进入被调函数后，实参变量将不再能使用。
- 在函数调用时，数据传送是单向的从实参传送给形参，而不能把形参的值反向地传送给实参。因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

下面举例讲解形参和实参在函数调用过程中的应用，程序示例如下：

51 单片机开发与应用技术详解

```
#include <stdio.h>                                //头文件
void sum(int n);                                  //函数声明

void main()                                       //主函数
{
    int n;                                       //定义变量
    n=4;                                         //变量初始化
    sum(n);                                     //调用函数
    printf("in the main function n=%d\n",n);    //输出主函数中的 n
}

void sum(int n)                                  //函数定义
{
    int i;                                       //定义变量
    for(i=n-1;i>0;i--)                          //for 循环计算累加值
        n=n+i;                                  //累加的结果赋值给 n
    printf("in the sum function n=%d\n",n);      //输出 sum 函数中 n 的值
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
in the sum function n=10
in the main function n=4
```

在本程序中，自定义了一个函数 **sum**，用于计算从 1~n 所有整数的和。在主函数中定义并初始化 **n** 值，并作为实参在调用时传送给 **sum** 函数的形参变量 **n**。注意，本例的形参变量和实参变量的标识符都为 **n**，但它们是两个不同的量，具有不同的作用域。从程序中可以看出，实参的值不随形参的变化而变化。

11.3.2 数组作为函数参数

数组可以作为函数的参数使用，进行数据传送。在 C51 语言中，数组用做函数参数有两种形式，一种是把数组元素（下标变量）作为实参使用；另一种是把数组名作为函数的形参和实参使用。下面分别进行讲解。

1. 数组元素作为函数实参

数组元素作为函数实参是指在发生函数调用时，把作为实参的数组元素的值传送给形参，实现单向的值传送。其中数组元素就是下标变量，与普通变量是完全相同的。下面举例讲解数组元素作为函数实参在程序设计中的应用。这里自定义一个函数 **fun**，用于判别一个整数数组中各元素的值，若是奇数则输出该值，否则不输出。程序代码示例如下：

```
#include <stdio.h>                                //头文件
void fun(int a)                                   //定义函数
{
    if(a%2==1)                                    //判断是否为奇数
        printf("%d\n",a);                        //如果是则输出
}

void main()                                       //主函数
{
    int i;
    int num[5]={1,2,3,4,5};                      //定义并初始化数组
    for(i=0;i<5;i++)                             //循环调用函数
```

```

{
    fun(num[i]);
}
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

1
3
5

```

在本程序中，首先定义一个无返回值函数 `fun`，并说明其形参 `a` 为整型变量。在函数体中根据 `a` 值输出相应的结果。在 `main` 函数中，首先初始化一个整型数组，然后用一个 `for` 语句循环 `fun` 函数，即把数组元素 `num[i]` 的值传送给形参 `a`，供 `fun` 函数使用。

2. 数组名作为函数参数

数组名作为函数参数是指在发生函数调用时，把作为实参的数组名传送给形参，实现单向的值传送。

在普通变量或数组元素作函数参数时，形参变量和实参变量是由编译系统分配的两个不同的内存单元。在函数调用时发生的值传送是把实参变量的值赋予形参变量；而在用数组名作为函数参数时，不是把实参数组的每一个元素的值都赋予形参数组的各个元素，只是将实参数组的首地址赋予形参数组名。形参数组名取得该首地址之后，也就等于有了实在的数组。实际上是形参数组和实参数组为同一数组，共同拥有一段内存空间。

下面举例讲解数组名作为函数参数的应用，定义一个数组 `a` 中存放了一个学生 5 门课程的成绩，求平均成绩。程序示例如下：

```

#include <stdio.h>                                //头文件
int sum(int num[5])                                //定义函数，用于求和
{
    int i;
    int sum=0;
    for(i=0;i<5;i++)
        sum=sum+num[i];
    return sum;
}

void main()                                        //主函数
{
    int sum1;
    int num[5]={1,2,3,4,5};                        //定义并初始化数组
    sum1=sum(num);                                  //调用函数
    printf("the sum of num is %d\n",sum1);          //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

the sum of num is 15

```

本例中，首先定义了一个整型函数 `sum`，有一个形参为整型数组 `num`，长度为 5。在函数 `sum` 中，把各元素值相加求和，并将结果返回给主函数。主函数中首先初始化一个数组 `num` 用于存放一个学生 5 门课程的成绩，然后以 `num` 作为实参调用 `sum` 函数，函数返回值保存在 `sum1` 种，最后输出 `sum1` 值。

注意：用数组元素作为实参时，只要数组类型和函数的形参变量的类型一致，并不要求函数的形参也是下标变量；而用数组名作函数参数时，则要求形参和相对应的实参都必须是类型相同的数组，都必须有明确的数组说明。当形参和实参二者不一致时，即会发生错误。

11.3.3 多维数组作为函数参数

除了一维数组外，多维数组也可以作为函数的参数。示例如下：

```
int sum(int a[3][4])
```

这里采用一个二维数组作为函数的形参，在函数定义时对形参数组可以指定每一维的长度。在 C51 语言中，也可省去第一维的长度。因此，以下写法也是合法的。

```
int sum(int a[][4])
```

下面举例讲解多维数组作为函数参数的应用，程序示例如下：

```
#include <stdio.h> //头文件
int sum(int num[3][4]) //定义函数
{
    int i,j;
    int sum=0;
    for(i=0;i<3;i++) //循环累加数组元素
    {
        for(j=0;j<4;j++)
            sum=sum+num[i][j];
    }
    return sum; //返回累加值
}

void main() //主函数
{
    int sum1;
    int num[3][4]={ {7,8,9},{4,5,6},{1,2,3}}; //定义并初始化二维数组
    sum1=sum(num); //调用函数
    printf("the sum of num[3][4] is %d\n",sum1); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
the sum of num[3][4] is 45
```

在本程序中，声明了一个函数 sum，其形参为二维整型数组，用于计算数组每个元素的和并返回。主程序中声明并初始化一个二维整型数组，并以数组名作为实参传递给函数 sum，最后输出计算结果。

11.3.4 指针作为函数参数

指针作为函数参数是指在发生函数调用时，把指针变量传送给形参，实现单向的值传送。使用指针类型作为函数的参数，实际向函数传递的是变量的地址。下面举例讲解指针作为函数参数的应用，程序示例如下：

```
#include <stdio.h> //头文件
void change(char *c) //定义函数，用于将小写字符改为大写字符
{
    if(*c>='a' & *c<='z') //判断，如果为小写，则改为大写字符
        *c=*c-32;
}

void main() //主函数
{
    char *c; //定义字符型指针变量
    *c='a'; //赋值
```

```
change(c); //调用函数
printf("%c",*c); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

A

在本程序中，实参是字符型指针变量，形参也是字符型指针变量。主函数在调用函数 `change` 时，将字符指针 `c` 的地址作为实参传给 `change` 函数。`change` 函数对该地址中的数据进行操作，因此，该地址所对应的变量值便改变了，从而达到将小写字符改为大写字符的目的。

注意：由于函数中获得了所传递变量的地址，因此，当函数结束后，该地址中的数据已经改变。

11.4 函数的返回值

函数的返回值是指函数被调用之后，在函数体中执行完相应的程序，并最终返回给主调函数的值。对于函数的返回值（或称函数的值），需要注意如下几点说明。

➤ 函数的值只能通过 `return` 语句返回主调函数。`return` 语句的一般形式如下：

```
return 表达式；
```

或者如下：

```
return (表达式)；
```

该语句的功能是计算表达式的值，并返回给主调函数。

- 在函数中允许有多个 `return` 语句，但每次调用只能有一个 `return` 语句被执行，因此只能返回一个函数值。
- 函数返回值的类型和函数定义中函数的类型应保持一致。如果函数值的类型和 `return` 语句中返回的值的类型不一致，则以函数类型为准自动进行类型转换，即函数类型决定返回值的类型。
- 如函数值为整型，在函数定义时可以省去类型说明。
- 使用类型说明符“`void`”来声明不返回函数值的函数，即其返回为“无值型”。一旦函数被定义为空类型后，就不能在主调函数中使用被调函数的函数值了。例如，在定义函数 `sum` 为空类型后，在主函数中，下述语句是错误的。

```
sum1=sum(n)；
```

一般来说，为了使程序有良好的可读性并减少出错，凡不要求返回值的函数都应定义为空类型。

11.5 函数调用

函数调用是指函数在主调函数中的调用形式。在前面的实例中其实已经用到了函数调用。在 C51 语言中，函数调用的一般形式如下：

函数名（实参列表）

其中，函数名即被调用的函数，实参列表是主调函数传递给被调函数的数据。通常，函数可以有三种调用方式。

➤ 函数语句：把函数作为一个语句，主要用于无返回值的函数。示例如下：

```
delay()；
```

➤ 函数表达式：函数出现在表达式中，主要用于有返回值的函数，将返回值赋值给变量。

示例如下：

```
c=min(x,y); //函数 min 求 x、y 中的最小值
➤ 函数参数：函数作为另一个函数的实参，主要用于函数的嵌套调用。示例如下：
c=min(x,min(y,z)); //函数 min 求 x、y、z 中的最小值
```

下面重点介绍一些经常使用的函数调用形式。

11.5.1 赋值调用与引用调用

赋值调用与引用调用是 C51 语言中最常用的参数传递方式，下面分别进行介绍。

- “赋值调用”(call by value)，这种方式中函数的形参是数值变量，函数调用时把参数的值复制到函数的形式参数中。赋值调用不会影响主调函数中的变量的数值。
- “引用调用”(call by reference)，这种方式中函数的形参是指针，函数调用时是把参数的地址复制给形式参数。在函数中，这个地址用来访问调用中所使用的实际参数。引用调用将会影响主调函数中的变量的数值。

一般来说，C51 语言经常使用赋值调用来传递参数，因此一般不能改变调用时所用变量的值。程序示例如下：

```
#include <stdio.h> //头文件
int mul(int x); //函数声明

void main( ) //主函数
{
    int t=10;
    printf("mul(%d)=%d\n",t,mul(t)); //函数赋值调用，并输出
}
int mul(int x) //定义函数
{
    x=x*x; //计算平方
    return(x); //返回结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
mul(10)=100
```

在本程序中，声明了函数 mul 用来计算平方值。传递给函数 mul 的参数值是复制给形式参数 x 的，当赋值语句 x =x*x 执行时，仅修改了函数内部的局部变量 x 的值。主函数中的变量 t 仍然为 10。

11.5.2 递归调用

递归调用是一个函数在它的函数体内调用它自身的函数调用方式。这种函数也称为“递归函数”。在递归函数中，主调函数又是被调函数。执行递归函数将反复调用其自身。每调用一次就进入新的一层。

示例如下：

```
int f(int x) //函数定义
{
    int y;
    z=f(y); //递归调用
    return z;
}
```

在本例中声明了一个函数 f，在函数体中同样调用了该函数，因此这是一个递归函数。该

函数在运行中将无休止地调用其自身，这当然是不正确的。

递归调用一定要防止无终止地进行调用，否则将是一个死循环，因此必须在函数内有终止递归调用的手段。在 C51 语言中，经常采用条件判断语句，满足某种条件后则停止递归调用，然后逐层返回。

从变量及内存分配角度来看，当函数调用自身的时候，将在堆栈中创建新的局部变量并分配内存，函数使用这些新的变量重新运行。当每次递归调用返回时，老的局部变量和参数就从堆栈中消除，从函数内的此次函数调用点重新启动运行。

最典型的采用递归调用的是计算 $n!$ 的值。 $n!$ 可用下述公式表示。

$$n! = 1; \quad (n=0, 1)$$

$$n! = n \times (n-1) \times \cdots \times 1; \quad (n > 1)$$

按照上面的公式，采用递归函数来进行计算，程序示例如下：

```
#include <stdio.h>                                //头文件
long nn(int n)                                     //定义函数
{
    long f;
    if(n>1)
        f=n*nn(n-1);                               //递归调用
    else
        f=1L;
    return f;
}

void main()                                         //主函数
{
    int num;
    long y;
    num=6;                                          //赋初值
    y=nn(num);                                     //调用函数计算阶乘
    printf("6!=%ld\n",y);                          //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
6!=720
```

在本程序中，递归函数 `nn` 用于计算 $n!$ 的值。主函数调用函数 `nn` 后即进入函数体执行，如果 n 不大于 1 将结束函数的执行，否则就递归调用函数 `nn` 自身。由于每次递归调用的实参为 $n-1$ ，即把 $n-1$ 的值赋予形参 n ，最后当 $n-1$ 的值为 1 时再作递归调用，形参 n 的值也为 1，将使递归终止，然后可逐层返回，便可以得到最终结果。

除了递归方法，本例也可以采用 `for` 循环语句来实现，即从 1 开始乘以 2，再乘以 3...直到 n 。`for` 循环语句比递归法更容易理解和实现。但是有些问题，例如 Hanoi 塔问题，则只能用递归算法才能实现。

采用递归函数可以把算法写得清晰简洁，而且某些问题，特别是与人工智能有关的问题，更适宜用递归方法。但大部分递归例程没有明显地减少代码规模和节省内存空间。递归形式比非递归形式运行速度要慢一些。这是因为附加的函数调用增加了时间开销。但在许多情况下，速度的差别不太明显。

！注意：编写递归函数时，必须使用 `if` 语句强制函数在未执行递归调用前返回。如果不这样做，在调用函数后，它永远不会返回，这是一个很容易犯的错误。

11.5.3 嵌套调用

嵌套调用即在被调函数中又调用了其他函数的调用形式。C51 语言中不允许进行嵌套的函数定义。但是允许调用其他的函数，因为除了主函数外，各个函数都是平行的。这与其他高级语言的子程序嵌套的情形是类似的。示例如下：

```
void main()                                //主函数
{
    .....
    Fun();                                //调用函数 Fun
    .....
}
void Fun()                                  //定义函数 Fun
{
    .....
    Fun1();                              //调用函数 Fun1
    .....
}
void Fun1()                                //定义函数 Fun1
{
    .....
}
```

在本程序中，首先执行 main 函数，当遇到调用 Fun 函数的语句时，即转去执行 Fun 函数。执行 Fun 函数时又遇到调用 Fun1 函数，从而又转去执行 Fun1 函数。Fun1 函数执行完毕返回 Fun 函数的断点处继续执行，Fun 函数执行完毕返回主函数的断点处继续执行。

这里举例讲解嵌套调用在程序设计中的应用。例如，计算 $s=(5 \% 2)^2+(7 \% 2)^2$ 的值。根据该数学表达式，可以编写两个函数，一个是用来计算平方值的函数 mul，另一个是用来计算余数的函数 ff。主函数先调用函数 ff 计算出余数，然后在函数 mul 中计算平方值，最后返回主函数。程序示例如下：

```
#include <stdio.h>                        //头文件
int mul(int k)                             //定义平方函数
{
    int a;
    a=k*k;                                //计算平方
    return a;                             //返回数值
}
int ff(int n)                              //定义函数
{
    int k,m;
    k=n % 2;                             //计算余数
    m=mul(k);                             //调用 mul 函数
    return m;                             //返回数值
}

void main()                                //主函数
{
    int i=5,j=7;
    int d;
    d=ff(i);                              //调用函数
    d+=ff(j);                             //调用函数
```

```
printf("%d\n",d);           //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

2

在本程序中，主函数之前首先定义了函数 `mul` 和 `ff`，其类型均为长整型。在主函数中，调用函数 `ff` 求值。在函数 `ff` 中计算余数后又调用函数 `mul`，在函数 `mul` 中完成最后的计算。函数 `mul` 执行完毕后把结果返回给函数 `ff`，再由函数 `ff` 返回主函数实现计算结果。

11.6 函数及其变量的作用域

函数及其变量的作用域是指函数的有效范围，以及函数内部变量的有效范围。下面分别进行讲解。

11.6.1 函数的作用域

函数的作用域是函数内部代码和数据的使用范围。在 C51 语言中，每一个函数都是一个独立的代码块，函数代码隐藏于函数内部，归该函数所有。使用函数时需要注意如下几点。

- 在程序中除了对函数的调用以外，其他任何函数中的任何语句都不能访问它。例如，使用跳转语句 `goto`，也只能在本函数内部使用，而不能从一个函数跳进其他函数内部执行。
- 定义在一个函数内部的数据无法与定义在另一个函数内部的数据进行作用，除非使用全程变量。
- C51 语言中所有函数的作用域都处于同一级别上，不可以在一个函数内再说明或定义另一个函数。
- C51 语言中一个函数对其他函数的调用是全程的，对于整个工程都是“可见”的，即使两个函数在不同的文件中，也不必附加任何说明语句而被另一函数调用。

11.6.2 函数的变量作用域

函数的变量作用域是指函数内部定义的变量的使用范围。C51 中的变量是可以在各个层次的函数开头处加以说明的，它的作用域仅在本函数内，当函数执行完毕后，变量及其分配的内存将消失。由于作用域的不同，一个函数中的变量即使与另一个函数中的变量名字相同，它们之间也是互不干扰的。下面举例讲解函数的变量作用域，程序示例如下：

```
#include <stdio.h>           //头文件
void Fun1(int n)             //定义函数 Fun1
{
    n=n+10;                  //重新赋值 n
    printf("in Fun1 n=%d\n",n); //输出 Fun1 中的 n
}
void Fun2(int n)             //定义函数 Fun2
{
    n=n*n;                   //重新赋值 n
    printf("in Fun2 n=%d\n",n); //输出 Fun2 中的 n
}
void main()                  //主函数
{
    int n=3;
    printf("in main n=%d\n",n); //输出主函数中的 n
    Fun1(n);                 //函数调用
```

51 单片机开发与应用技术详解

```
Fun2(n);
printf("in main n=%d\n",n);           //输出主函数中的 n
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
in main n=3
in Fun1 n=13
in Fun2 n=9
in main n=3
```

在本程序中，主函数在调用其他函数之前首先赋值 `n=3`，接着调用函数 `Fun1` 和 `Fun2`。在这两个函数中，分别输出函数内部的 `n` 值。由于不同函数中的变量由于作用域不同，因此其值互不干扰。

11.7 main 函数

`main` 函数是 C51 语言程序中的特殊函数，是整个程序的入口。一般来说，一个 C51 语言程序都要有一个主函数 `main`，而且只能有一个。C51 语言程序的执行总是从 `main` 函数开始，如果有其他函数，则完成对其他函数的调用后再返回到主函数，最后由 `main` 函数结束整个程序。


一个 C51 语言程序常由一个主函数和若干个其他函数构成。由主函数调用其他函数，其他函数之间也可以相互调用。`main` 函数作为主调函数允许 `main` 调用其他函数并传递参数。`main` 函数既可以是无参函数，也可以是有参的函数。对于有参的形式来说，就需要向其传递参数。但是其他任何函数均不能调用 `main` 函数。当然也同样无法向 `main` 函数传递参数，因此 `main` 函数的参数只能由程序之外传递而来。

带参数的 `main` 函数的形式如下：

```
int main(int argc,char *argv[])
```

其中，主函数的参数包含一个整型和一个指针数组。整型参数是表示被调用函数所带命令行的参数的数目；指针数组参数中的每个元素是指向包含命令行参数的指针，即每个指针对应一个字符串，而第一个指针通常指向命令名字符串。`main` 函数的返回值，主要用于向系统返回一个运行状态码。

带参数的 `main` 函数主要用于复杂应用程序的过程间通信，对于一般的单一程序是不起任何作用的。因此，一般采用无参的形式即可，如 `void main()`。

 **注意：**对于实时多任务操作系统 RTX-51 程序，可以没有 `main` 主函数，它是从第 0 个任务开始执行的。这将在后面的章节介绍。

11.8 小结

本章详细介绍了 C51 语言中函数的使用，包括函数的定义、形参和实参、函数的返回值、函数的调用、函数及其变量的作用域。本章还结合单片机程序设计的特点，介绍了 C51 程序中的 `main` 函数。函数是 C51 语言中的重要概念，灵活运用函数可以实现程序的模块化设计。因此，熟练掌握本章内容，是 C51 语言程序设计的基础。

第 12 章 C51 语言的常用库

函数详解

C51 语言的编译器中包含有丰富的库函数，使用库函数可以大大简化用户程序设计的工作量，提高编程效率。每个库函数都在相应的头文件中给出了函数原型声明，在使用时，必须在源程序的开始处使用预处理命令 `#include` 将有关的头文件包含进来。

C51 语言库函数中类型的选择考虑到了 8051 单片机的结构特性，用户在自己的应用程序中应尽可能地使用最小的数据类型，以最大限度地发挥 8051 单片机的性能，同时可减少应用程序的代码长度。下面将 C51 语言的库函数分类列出并详细介绍其用法。

12.1 字符函数

字符函数的原型声明包含在头文件 `CTYPE.H` 中。下面介绍常用的一些字符函数。

12.1.1 检查英文字母函数

检查英文字母函数用于检查形参字符是否为英文字母，其函数原型如下：

```
bit isalpha(char c);
```

其中，`c` 为待判断的字符，如果是英文字母则返回 1，否则返回 0。程序示例如下：

```
#include<ctype.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    unsigned char i;                               //变量声明
    char *p;
    SCON=0x50;                                     //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器
    for (i=0;i<128;i++)                             //循环输出
    {
        p=(isalpha(i)? "YES":"NO");               //调用 isalpha 函数
        printf("isalpha(%c)=%s\n",i,p);           //输出结果
    }
}
```


该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 `isalpha` 函数来判断 128 个字符对应的是否为英文字母，运行的结果将通过串口 0 输出。

12.1.2 检查字母数字函数

检查字母数字函数用于检查形参字符是否为英文字母或数字字符，其函数原型如下：

```
bit isalnum(char c);
```

其中，`c` 为待判断的字符，如果是英文字母或数字字符则返回 1，否则返回 0。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
    unsigned char i; //变量声明
    char *p;
    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器
    for(i=0;i<128;i++) //循环输出
    {
        p=(isalnum(i)? "YES":"NO"); //调用 isalnum 函数
        printf("isalnum(%c)=%s\n",i,p); //输出结果
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 `isalnum` 函数来判断 128 个字符对应的是否为英文字母或数字，运行的结果将通过串口 0 输出。

12.1.3 检查控制字符函数

检查控制字符函数用于检查形参字符是否为控制字符，其函数原型如下：

```
bit iscntrl (char c);
```

其中，`c` 为待判断的字符。控制字符其取值范围为 0x00~0x1F 之间或等于 0x7F，如果是，则返回 1，否则返回 0。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
    unsigned char i; //变量声明
    char *p;
    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
```

```

    TH1=0xF4;
    IE|=0x90;                                //中断
    TR1=1;                                    //启动定时器
for(i=0;i<128;i++)                          //循环输出
{
    p=(iscntrl(i)? "YES":"NO");              //调用 iscntrl 函数
    printf("iscntrl(%c)=%s\n",i,p);          //输出结果
}
}

```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 iscntrl 函数来判断 128 个字符对应的是否为控制字符，运行的结果将通过串口 0 输出。

12.1.4 十进制数字检查函数

十进制数字检查函数用于检查形参字符是否为十进制数字，其函数原型如下：

```
bit isdigit (char c);
```

其中，c 为待判断的字符，如果是十进制数字则返回 1，否则返回 0。程序示例如下：

```

#include<ctype.h>                            //头文件
#include<stdio.h>
#include <reg51.h>
void main(void)                              //主函数
{
    unsigned char i;                          //变量声明
    char *p;
    SCON=0x50;                                //串口模式 1，允许接收
    TMOD|=0x20;                                //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                //设置 SMOD=1
    TL1=0xF4;                                  //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                  //中断
    TR1=1;                                    //启动定时器
for(i=0;i<128;i++)                          //循环输出
{
    p=(isdigit(i)? "YES":"NO");              //调用 isdigit 函数
    printf("isdigit(%c)=%s\n",i,p);          //输出结果
}
}

```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 isdigit 函数来判断 128 个字符对应的是否为十进制数字，运行的结果将通过串口 0 输出。

12.1.5 可打印字符检查函数

可打印字符检查函数用于检查形参字符是否为可打印字符，其函数原型如下：

```
bit isgraph (char c);
```

其中，c 为待判断的字符。可打印字符的取值范围为 0x21~0x7C，不包含空格，如果是可打印字符则返回 1，否则返回 0。程序示例如下：

```

#include<ctype.h>                            //头文件
#include<stdio.h>
#include <reg51.h>

```

```
void main(void) //主函数
{
    unsigned char i; //变量声明
    char *p;

    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    for(i=0;i<128;i++) //循环输出
    {
        p=(isgraph(i)? "YES":"NO"); //调用 isgraph 函数
        printf("isgraph(%c)=%s\n",i,p); //输出结果
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 isgraph 函数来判断 128 个字符对应的是否为可打印字符，运行的结果将通过串口 0 输出。

12.1.6 包含空格的可打印字符检查函数

包含空格的可打印字符检查函数用于检查形参字符是否为可打印字符及空格，其函数原型如下：

```
bit isprint (char c);
```

其中，c 为待判断字符。如果是则返回 1，否则返回 0。该函数与 isgraph 的区别在于包含了空格符，空格符为 0x20。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>

void main(void) //主函数
{
    unsigned char i; //变量声明
    char *p;

    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    for(i=0;i<128;i++) //循环输出
    {
        p=(isprint(i)? "YES":"NO"); //调用 isprint 函数
        printf("isprint(%c)=%s\n",i,p); //输出结果
    }
}
```

```

}
}

```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 `isprint` 函数来判断 128 个字符对应的是否为包含空格的可打印字符，运行的结果将通过串口 0 输出。

12.1.7 格式字符检查函数

格式字符检查函数用于检查形参字符是否为标点、空格或格式字符，其函数原型如下：

```
bit ispunct (char c);
```

其中，`c` 为待判断字符，如果是则返回 1，否则返回 0。程序示例如下：

```

#include<ctype.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    unsigned char i;                               //变量声明
    char *p;

    SCON=0x50;                                     //串口模式 1，允许接收
    TMOD|=0x20;                                     //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                         //启动定时器

    for(i=0;i<128;i++)                             //循环输出
    {
        p=(ispunct(i)? "YES":"NO");               //调用 ispunct 函数
        printf("ispunct(%c)=%s\n",i,p);           //输出结果
    }
}

```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 `ispunct` 函数来判断 128 个字符对应的是否为标点、空格或格式字符，运行的结果将通过串口 0 输出。符合函数 `ispunct` 检查的格式字符包含 `!、”、#、%、&、’、(、)、*、+、-、,、.、/、\、;、:、<、=、>、?、@、[、]、~、^、_、{、}、|`。

12.1.8 小写英文字母检查函数

小写英文字母检查函数用于检查形参字符是否为小写英文字母，其函数原型如下：

```
bit islower(char c);
```

其中，`c` 为待判断字符。如果是小写英文字母则返回 1，否则返回 0。程序示例如下：

```

#include<ctype.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    unsigned char i;                               //变量声明
    char *p;

```

```
SCON=0x50; //串口模式 1，允许接收
TMOD|=0x20; //初始化 T1 为定时功能，模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器

for(i=0;i<128;i++) //循环输出
{
    p=(islower(i)? "YES":"NO"); //调用 islower 函数
    printf("islower(%c)=%s\n",i,p); //输出结果
}
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 islower 函数来判断 128 个字符对应的是否为小写英文字母，运行的结果将通过串口 0 输出。

12.1.9 大写英文字母检查函数

大写英文字母检查函数用于检查形参字符是否为大写英文字母，其函数原型如下：

```
bit isupper(char c);
```

其中，c 为待判断字符。如果是大写英文字母则返回 1，否则返回 0。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
    unsigned char i; //变量声明
    char *p;
    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    for(i=0;i<128;i++) //循环输出
    {
        p=( isupper(i)? "YES":"NO"); //调用 isupper 函数
        printf("isupper(%c)=%s\n",i,p); //输出结果
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 isupper 函数来判断 128 个字符对应的是否为大写英文字母，运行的结果将通过串口 0 输出。

12.1.10 控制字符检查函数

控制字符检查函数用于检查形参字符是否为控制字符，其函数原型如下：

```
bit isspace (char c);
```

其中，c 为待判断字符。如果是控制字符则返回 1，否则返回 0。控制字符包括空格、制表

符、回车、换行、垂直制表符和送纸，其取值范围为 0x09~0x0d，或者为 0x20。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
    unsigned char i; //变量声明
    char *p;
    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器
    for(i=0;i<128;i++) //循环输出
    {
        p=(isspace(i)? "YES":"NO"); //调用 isspace 函数
        printf("isspace(%c)=%s\n",i,p); //输出结果
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 isspace 函数来判断 128 个字符对应的是否为控制字符，运行的结果将通过串口 0 输出。

12.1.11 十六进制数字检查函数

十六进制数字检查函数用于检查形参字符是否为十六进制数字，其函数原型如下：

```
bit isxdigit (char c);
```

其中，c 为待判断字符。如果是 16 进制数字字符则返回 1，否则返回 0。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
    unsigned char i; //变量声明
    char *p;
    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器
    for(i=0;i<128;i++) //循环输出
    {
        p=(isxdigit(i)? "YES":"NO"); //调用 isxdigit 函数
        printf("isxdigit(%c)=%s\n",i,p); //输出结果
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中对串口进行了初始化，接着循环调用 isxdigit 函数来判断 128 个字符对应的是否为 16 进制数字字符，运行的结果将通过串口 0 输出。

12.1.12 十六进制数字转换函数

十六进制数字检查函数用于转换形参字符为十六进制数字，其函数原型如下：

```
char toint (char c);
```

其中，c 为待转换字符。该函数将形参字符 0~9、a~f（大小写无关）转换为 16 进制数字。其中，对于字符 0~9，返回值为 0H~9H，对于 ASCII 字符 a~f（大小写无关），返回值为 0AH~0FH。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
void main(void) //主函数
{
    char ch; //变量声明
    unsigned long k;
        SCON=0x50; //串口模式 1，允许接收
        TMOD|=0x20; //初始化 T1 为定时功能，模式 2
        PCON|=0x80; //设置 SMOD=1
        TL1=0xF4; //波特率 4800bit/s，初值
        TH1=0xF4;
        IE|=0x90; //中断
        TR1=1; //启动定时器
        ch='2'; //字符变量赋值
        k= toint(ch); //调用 toint 函数
        printf("toint(%c)=%ld\n",ch,k); //输出结果
        ch='e'; //字符变量赋值
        k= toint(ch); //调用 toint 函数
        printf("toint(%c)=%ld\n",ch,k); //输出结果
    }
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
toint(2)=2
toint(e)=14
```

在该程序中首先对串口进行了初始化，接着为字符变量 ch 赋值，然后调用 toint 函数将字符转换为 16 进制数字，运行的结果将通过串口 0 输出。

12.1.13 大写字符转换函数

大写字符转换函数用于将大写字符转换为小写字符，其函数原型如下：

```
char tolower (char c);
```

其中，c 为待转换的大写字符。如果字符参数不在 A~Z 之间，则该函数将不起作用，而直接返回原字符。程序示例如下：

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
```

```

char ch;                                     //变量声明
char ch_lower;
    SCON=0x50;                             //串口模式 1，允许接收
    TMOD|=0x20;                             //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                             //设置 SMOD=1
    TL1=0xF4;                              //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                              //中断
    TR1=1;                                 //启动定时器
    ch='R';                                //字符变量赋值
    ch_lower= tolower(ch);                 //调用 tolower 函数
    printf("tolower(%c)=%c\n",ch,ch_lower); //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
tolower(R)=r
```

在该程序中首先对串口进行了初始化，接着为字符变量 `ch` 赋值，然后调用 `tolower` 函数将大写字符转换为小写字符，运行的结果将通过串口 0 输出。

12.1.14 小写字符转换函数

小写字符转换函数用于将小写字符转换为大写字符，其函数原型如下：

```
char toupper (char c);
```

其中，`c` 为待转换的小写字符。如果字符参数不在 `a~z` 之间，则该函数将不起作用，而直接返回原字符。程序示例如下：

```

#include<ctype.h>                           //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void)                             //主函数
{
    char ch;                               //变量声明
    char ch_upper;

    SCON=0x50;                             //串口模式 1，允许接收
    TMOD|=0x20;                             //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                             //设置 SMOD=1
    TL1=0xF4;                              //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                              //中断
    TR1=1;                                 //启动定时器
    ch='q';                                //字符变量赋值
    ch_upper= toupper(ch);                 //调用 toupper 函数
    printf("toupper(%c)=%c\n",ch,ch_upper); //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
toupper(R)=r
```

在该程序中首先对串口进行了初始化，接着为字符变量 `ch` 赋值，然后调用 `toupper` 函数将小写字符转换为大写字符，运行的结果将通过串口 0 输出。

12.1.15 ASCII 字符转换函数

ASCII 字符转换函数用于将任何字符型参数缩小到有效的 ASCII 范围之内,其函数原型如下:

```
char toascii (char c);
```

其中, c 为待转换的字符。该函数执行的操作是将形参数值和 0x7f 做与运算,从而去掉第 7 位以上的所有位数。如果形参已是有效的 ASCII 字符,则不作处理,直接返回原字符。程序示例如下:

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
int ch; //变量声明
char k;
SCON=0x50; //串口模式 1, 允许接收
TMOD|=0x20; //初始化 T1 为定时功能, 模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s, 初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器
ch=245; //字符变量赋值
k=toascii(ch); //调用 toascii 函数
printf("toascii(%d)=%c\n",ch,k); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行,运行的结果如下:

```
toascii(245)=u
```

在该程序中首先对串口进行了初始化,由于字符型变量可以表示为整型,因此变量 ch 赋值,然后调用 toascii 函数将其转换为有效的 ASCII 字符,运行的结果将通过串口 0 输出。

12.1.16 大写字符宏转换函数

大写字符宏转换函数用于大写字符转换为小写字符,其函数原型如下:

```
char _tolower(char c);
```

其中, c 为待转换的大写字符。这其实是一个由宏定义完成的操作,其功能是将字符参数 c 与常数 0x20 逐位进行或运算,从而将大写字符转换为小写字符。程序示例如下:

```
#include<ctype.h> //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void) //主函数
{
char ch; //变量声明
char ch_lower1;
char ch_lower2;
SCON=0x50; //串口模式 1, 允许接收
TMOD|=0x20; //初始化 T1 为定时功能, 模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s, 初值
TH1=0xF4;
```

```

    IE|=0x90;                                //中断
    TR1=1;                                    //启动定时器
    ch='R';                                    //字符变量赋值
    ch_lower1=tolower(ch);                    //调用 tolower 函数
    printf("tolower(%c)=%c\n",ch,ch_lower1); //输出结果
    ch_lower2=_tolower(ch);                  //调用 _tolower 函数
    printf("_tolower(%c)=%c\n",ch,ch_lower2); //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

tolower(R)=r
_tolower(R)=r

```

在该程序中首先对串口进行了初始化，接着为字符变量 `ch` 赋值，然后分别调用 `tolower` 函数和 `_tolower` 函数将大写字符转换为小写字符，运行的结果将通过串口 0 输出。从上面可见两者运行结果完全相同。

12.1.17 小写字符宏转换函数

小写字符宏转换函数用于小写字符转换为大写字符，其函数原型如下：

```
char _toupper (char c);
```

其中，`c` 为待转换的小写字符。这其实是一个由宏定义完成的操作，其功能是将字符参数 `c` 与常数 `0xdf` 逐位进行与运算，从而将小写字符转换为大写字符。程序示例如下：

```

#include<ctype.h>                                //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void)                                    //主函数
{
    char ch;                                       //变量声明
    char ch_upper1;
    char ch_upper2;
    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器
    ch='r';                                         //字符变量赋值
    ch_upper1=toupper(ch);                        //调用 toupper 函数
    printf("toupper(%c)=%c\n",ch,ch_upper1);      //输出结果
    ch_upper2=_toupper(ch);                      //调用 _toupper 函数
    printf("_toupper(%c)=%c\n",ch,ch_upper2);     //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

toupper(r)=R
_toupper(r)=R

```

在该程序中首先对串口进行了初始化，接着为字符变量 `ch` 赋值，然后分别调用 `toupper` 函数和 `_toupper` 函数将小写字符转换为大写字符，运行的结果将通过串口 0 输出。从上面可见两者运行结果完全相同。

12.2 字符串函数

字符串函数的原型声明包含在头文件 **STRINGH** 中。在 C51 语言中，字符串应包括两个或多个字符，字符串的结尾以空字符来表示。字符串函数通过字符串指针来对字符串进行处理。下面介绍常用的字符串函数。

12.2.1 字符查找函数

字符查找函数用于在字符串中顺序查找字符，其函数原型如下：

```
void *memchr(void*s1,char val,int len);
```

其中，s1 为输入字符串，val 为待查找的字符，len 为查找的长度范围。该函数的功能是在字符串 s1 中顺序搜索前 len 个字符以找出字符 val，如果找到则返回 s1 中指向 val 的指针，如果没有找到则返回 NULL。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>
voidmain(void)                                     //主函数
{
    static char str[20]="This is a test!";
    void *c;
        SCON=0x50;                                //串口模式 1，允许接收
        TMOD|=0x20;                                //初始化 T1 为定时功能，模式 2
        PCON|=0x80;                                //设置 SMOD=1
        TL1=0xF4;                                  //波特率 4800bit/s，初值
        TH1=0xF4;
        IE|=0x90;                                  //中断
        TR1=1;                                     //启动定时器
    c=memchr(str,'a',sizeof(str));                 //调用 memchr 函数
    if(c==NULL)                                    //未找到字符
        printf("'a' was not found in the str\n");
    else                                           //找到字符
        printf("found 'a' in the str\n");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
found 'a' in the str
```

在该程序中，首先初始化了一个字符串 str，并对串口进行了初始化，接着调用 memchr 函数搜索字符 a，运行的结果将通过串口 0 输出。

12.2.2 指定长度的字符串比较函数

指定长度的字符串比较函数用于按照指定的长度比较两个字符串的大小，其函数原型如下：

```
char memcmp(void*s1, void*s2,int len);
```

其中，s1 和 s2 为输入字符串，len 为比较的长度。该函数的功能是逐个比较字符串 s1 和 s2 的前 len 个字符，如果相等则返回 0，如果字符串 s1 大于 s2，则返回一个正数，如果字符串 s1 小于 s2，则返回一个负数。如果两个字符串的长度小于 len，该函数仍将一直比较 len 个字符，这种情况下，有可能结果是错误的。因此应该保证 len 不能超过最短字符串的长度。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
```

```

#include <reg51.h>

void main(void)                                //主函数
{
    static char str1[]="012345abc";
    static char str2[]="012345ABC";
    char i;

    SCON=0x50;                                //串口模式 1，允许接收
    TMOD|=0x20;                                //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                //设置 SMOD=1
    TL1=0xF4;                                  //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                  //中断
    TR1=1;                                     //启动定时器

    i=memcmp(str1, str2,16);                  //调用 memcmp 函数
    if(i<0)
        printf("str1<str2\n");
    else if(i>0)
        printf("str1>str2\n");
    else
        printf("str1==str2\n");
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str1>str2
```

在该程序中，首先初始化了字符串 str1 和 str2，并对串口进行了初始化，接着调用 memcmp 函数比较这两个字符串，运行的结果将通过串口 0 输出。

12.2.3 字符串复制函数

字符串复制函数用于复制指定长度的字符串，其函数原型如下：

```
void*memcpy(void*dest,void*src,int len);
```

其中，dest 为目标字符串，src 为源字符串，len 为复制的长度。该函数的功能是从 src 所指向的字符串中复制 len 个字符到 dest 字符串中，其返回值指向 dest 中的最后一个字符的指针。程序示例如下：

```

#include<string.h>                            //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                                //主函数
{
    static char str1[]="012345abc";
    static char str2[20];
    char *p;

    SCON=0x50;                                //串口模式 1，允许接收
    TMOD|=0x20;                                //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                //设置 SMOD=1
    TL1=0xF4;                                  //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                  //中断
    TR1=1;                                     //启动定时器
}

```

```
p=memcpy(str2,str1,sizeof(str1));           //调用 memcpy 函数
printf("str2=%s\n",str2);                   //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str2=012345abc
```

在该程序中首先初始化了字符串 str1，并对串口进行了初始化，接着调用 memcpy 函数将字符串 str1 复制到字符串 str2 中，运行的结果将通过串口 0 输出。

12.2.4 带终止字符的字符串复制函数

带终止字符的字符串复制函数用于复制字符串，如果遇到终止字符则停止复制，其函数原型如下：

```
void*memccpy(void*dest, void*src,char val,int len);
```

其中，dest 为目标字符串，src 为源字符串，val 为终止字符，len 为复制的长度。该函数的功能是复制字符串 src 中的 len 个字符到 dest 中，复制 len 个字符后则返回 NULL。如果遇到字符 val 则停止复制，此时返回一个指向 dest 中的下一个元素的指针。程序示例如下：

```
#include<string.h>           //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)               //主函数
{
    static char str1[]="This is a test!"; //初始化
    static char str2[20];
    char *p;

    SCON=0x50;               //串口模式 1，允许接收
    TMOD|=0x20;              //初始化 T1 为定时功能，模式 2
    PCON|=0x80;              //设置 SMOD=1
    TL1=0xF4;                //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                //中断
    TR1=1;                   //启动定时器

    p=memccpy(str2,str1,'a',sizeof(str1)); //调用 memccpy 函数
    printf("str2=%s\n",str2);             //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str2=This is a
```

在该程序中，首先初始化了字符串 str1，并对串口进行了初始化，接着调用 memccpy 函数将字符串 str1 复制到字符串 str2 中，如果遇到字符 a 则停止复制，运行的结果将通过串口 0 输出。

12.2.5 字符串移动函数

字符串移动函数同样用于复制字符串，其函数原型如下：

```
void *memmove(void*dest,void*src,int len);
```

其中，dest 为目标字符串，src 为源字符串，len 为复制长度。该函数的功能是从 src 所指向的字符串中复制 len 个字符到 dest 字符串中，其返回值指向 dest 中的最后一个字符的指针。

其功能与 `memcpy` 相同，但是复制区间 `src` 与 `dest` 可以发生交迭。程序示例如下：

```
#include<string.h>           //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)               //主函数
{
    static char str[]="this is line 1 " //初始化字符串
    "this is line2 "
    "this is line 3 ";

    SCON=0x50;               //串口模式 1，允许接收
    TMOD|=0x20;              //初始化 T1 为定时功能，模式 2
    PCON|=0x80;              //设置 SMOD=1
    TL1=0xF4;                //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                //中断
    TR1=1;                   //启动定时器

    printf("str before=%s\n",str);
    memmove(&str[0],&str[15],32); //调用 memmove 函数
    printf("str after=%s\n",str);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str before=this is line1 this is line2 this is line3
str after=this is line2 this is line3
```

在该程序中首先初始化了字符串 `str`，并对串口进行了初始化，接着调用 `memmove` 函数移动字符串，运行的结果将通过串口 0 输出。

12.2.6 字符串填充函数

字符串填充函数用于按规定的字符填充字符串，其函数原型如下：

```
void*memset(void*s,char val,int len);
```

其中，`s` 为待填充的字符串，`val` 为填充字符，`len` 为填充的长度。该函数实现的操作是用字符 `val` 来填充字符串 `s`，共填充 `len` 个单元。程序示例如下：

```
#include<string.h>           //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)               //主函数
{
    static char str[]="this is line 1 ";

    SCON=0x50;               //串口模式 1，允许接收
    TMOD|=0x20;              //初始化 T1 为定时功能，模式 2
    PCON|=0x80;              //设置 SMOD=1
    TL1=0xF4;                //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                //中断
    TR1=1;                   //启动定时器
```

51 单片机开发与应用技术详解

```
printf("str before=%s\n",str);
memset(str,'\0',sizeof(str));           //调用 memset 函数
printf("str after=%s\n",str);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str before=this is line 1
str after=
```

在该程序中，首先初始化了字符串 str，并对串口进行了初始化，接着调用 memset 函数将字符串全部填充为空字符“\0”，运行的结果将通过串口 0 输出。

12.2.7 字符串追加函数

字符串追加函数用于复制字符串到另一个字符串的尾部，其函数原型如下：

```
void *strcat(char *s1,char *s2);
```

其中，s1 为目标字符串，s2 为待复制的字符串。该函数实现的操作是将字符串 s2 复制到字符串 s1 的尾部。其中字符串 s1 要有足够的长度来保存两个字符串。该函数的返回值指向字符串 s1 中的第一个字符的指针。程序示例如下：

```
#include<string.h>           //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)               //主函数
{
    static char str1[50]="this is line 1 ";
    static char str2[]="this is line 2 ";

    SCON=0x50;               //串口模式 1，允许接收
    TMOD|=0x20;              //初始化 T1 为定时功能，模式 2
    PCON|=0x80;              //设置 SMOD=1
    TL1=0xF4;                //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                //中断
    TR1=1;                   //启动定时器

    strcat(str1,str2);        //调用 strcat 函数
    printf("str1=%s\n",str1);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str1=this is line 1 this is line 2
```

在该程序中，首先初始化了字符串 str1 和 str2，并对串口进行了初始化，接着调用 strcat 函数将字符串 str2 复制到字符串 str1 的尾部，运行的结果将通过串口 0 输出。

12.2.8 指定长度的字符串追加函数

指定长度的字符串追加函数用于复制指定长度的字符串到另一个字符串的尾部，其函数原型如下：

```
void*strncat(,char *s1,char *s2,int n);
```

其中，s1 为目标字符串，s2 为待复制的字符串，n 为复制的长度。该函数实现的操作是从字符串 s2 中复制 n 个字符添加到字符串 s1 的尾部。其中，如果字符串 s2 的长度比 n 小，则将全部复制字符串 s2（包括串结束符）。程序示例如下：

```

#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    static char str1[50]="this is line 1 ";
    static char str2[]="this is line 2 ";

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                      //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                      //中断
    TR1=1;                                         //启动定时器

    strncat(str1,str2,7);                          //调用 strncat 函数
    printf("str1=%s\n",str1);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str1=this is line 1 this is
```

在该程序中，首先初始化了字符串 `str1` 和 `str2`，并对串口进行了初始化，接着调用 `strncat` 函数从字符串 `str2` 中复制 7 个字符到字符串 `str1` 的尾部，运行的结果将通过串口 0 输出。

12.2.9 字符串比较函数

字符串比较函数用于比较两个字符串的大小，其函数原型如下：

```
char strcmp(char *s1,char *s2);
```

其中，`s1` 和 `s2` 为待比较的字符串。该函数的功能是比较字符串 `s1` 和 `s2`，如果两者相等则返回 0；如果 `s1<s2`，则返回一个负数；如果 `s1>s2`，则返回一个正数。程序示例如下：

```

#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    static char str1[]="012345abc";
    static char str2[]="012345abcd";
    char i;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                      //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                      //中断
    TR1=1;                                         //启动定时器

    i=strcmp(str1, str2);                          //调用 strcmp 函数
    if(i<0)                                        //输出比较结果
        printf("str1<str2\n");
}

```


51 单片机开发与应用技术详解

```
else if(i>0)
    printf("str1>str2\n");
else
    printf("str1==str2\n");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str1<str2
```

在该程序中，首先初始化了字符串 str1 和 str2，并对串口进行了初始化，接着调用 strcmp 函数比较字符串 str1 和 str2，运行的结果将通过串口 0 输出。

12.2.10 包含结束符的字符串比较函数

包含结束符的字符串比较函数用于比较两个字符串的大小，其函数原型如下：

```
char*strncmp(char *s1,char *s2,int n);
```

其中，s1 和 s2 为待比较的字符串，n 为比较的长度。该函数的功能是比较字符串 s1 和 s2 的前 n 个字符，如果两者相等则返回 0；如果 s1<s2，则返回一个负数；如果 s1>s2，则返回一个正数。这里需要和 memcmp 函数相区别，如果字符串的长度小于 n，则 strncmp 函数比较到字符串结束符后便停止，这和 memcmp 函数是不一样的。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                     //主函数
{
    static char str1[]="012345abc";
    static char str2[]="012345abcd";
    char i;

    SCON=0x50;                                     //串口模式 1，允许接收
    TMOD|=0x20;                                     //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    i=strncmp(str1, str2,20);                       //调用 strncmp 函数
    if(i<0)                                         //输出比较结果
        printf("str1<str2\n");
    else if(i>0)
        printf("str1>str2\n");
    else
        printf("str1==str2\n");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str1<str2
```

在该程序中，首先初始化了字符串 str1 和 str2，并对串口进行了初始化，接着调用 strncmp 函数比较字符串 str1 和 str2，运行的结果将通过串口 0 输出。其中，strncmp 函数中参数 20 限制仅比较 20 个字符，而字符串长度不够 20 个，因此，比较到字符串结束符便停止比较。

12.2.11 字符串覆盖函数

字符串覆盖函数用于将一个字符串覆盖另一个字符串，其函数原型如下：

```
char* strcpy(char *s1,char *s2);
```

其中, `s1` 为目标字符串, `s2` 为源字符串。该函数的功能是将字符串 `s2` (包括结束符) 复制到字符串 `s1` 中的第 1 个字符指针处。这里需要注意和 `strcat` 函数相区别, `strcat` 函数将字符串 `s2` 复制到字符串 `s1` 的末尾。程序示例如下:

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                     //主函数
{
    static char str1[50]="this is line 1 ";
    static char str2[]="this is line 2 ";

    SCON=0x50;                                     //串口模式 1, 允许接收
    TMOD|=0x20;                                     //初始化 T1 为定时功能, 模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s, 初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                         //启动定时器

    strcpy(str1,str2);                             //调用 strcpy 函数
    printf("str1=%s\n",str1);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行, 运行的结果如下:

```
str1=this is line 2
```

在该程序中, 首先初始化了字符串 `str1` 和 `str2`, 并对串口进行了初始化, 接着调用 `strcpy` 函数将字符串 `str2` 复制到 `str1` 的前面覆盖原字符串, 运行的结果将通过串口 0 输出。

12.2.12 指定长度的字符串覆盖函数

指定长度的字符串覆盖函数用于将一个指定长度的字符串覆盖另一个字符串, 其函数原型如下:

```
char*strncpy(char *s1,char *s2,int n);
```

其中, `s1` 为目标字符串, `s2` 为源字符串, `n` 为长度。该函数实现的操作是从字符串 `s2` (包括结束符) 中复制 `n` 个字符到字符串 `s1` 中的第一个字符指针处。如果字符串 `s2` 的长度小于 `n`, 则 `s1` 串以 0 补齐到长度 `n`。程序示例如下:

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                     //主函数
{
    static char str1[50]="this is line 1 ";
    static char str2[]="there is line 2 ";

    SCON=0x50;                                     //串口模式 1, 允许接收
    TMOD|=0x20;                                     //初始化 T1 为定时功能, 模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s, 初值
    TH1=0xF4;
```

51 单片机开发与应用技术详解

```
IE|=0x90; //中断
TR1=1; //启动定时器

strncpy(str1,str2,5); //调用 strncpy 函数
printf("str1=%s\n",str1);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
str1=thereis line 1
```

在该程序中，首先初始化了字符串 str1 和 str2，并对串口进行了初始化，接着调用 strncpy 函数从字符串 str2 中复制 5 个字符到 str1 的前面覆盖原字符串，运行的结果将通过串口 0 输出。

12.2.13 获取字符串个数函数

获取字符串个数函数用于返回字符串中字符总数，其函数原型如下：

```
char*strlen(char *s1);
```

其中，s1 为输入字符串。该函数的功能是获取字符串 s1 中的字符个数，返回值的大小不包括结尾的字符串结束符。程序示例如下：

```
#include<string.h> //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void) //主函数
{
static char str[50]="this is line 1 ";
int i;

SCON=0x50; //串口模式 1，允许接收
TMOD|=0x20; //初始化 T1 为定时功能，模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器

i=strlen(str); //调用 strlen 函数
printf("strlen(str)=%d\n",i); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
strlen(str)=15
```

在该程序中，首先初始化了字符串 str，并对串口进行了初始化，接着调用 strlen 函数获取该字符串的长度，运行的结果将通过串口 0 输出。

12.2.14 搜索字符串函数

搜索字符串函数用于搜索字符串出现的位置，其函数原型如下：

```
char*strstr(const char *s1,char*s2);
```

其中，s1 为目标字符串，s2 为搜索的字符串。该函数实现的操作是在字符串 s1 中搜索第一次出现字符串 s2 的位置，并返回该处的指针。如果字符串 s1 中不包括字符串 s2，则该函数返回一个空指针。程序示例如下：

```

#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    static char str[50]="this is line 1 ";
    char * p;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    p=strstr(str,"line");                          //调用 strstr 函数
    printf("p=strstr(str,\"is\")=%s\\n\",p);        //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
p=strstr(str,"line")=line 1
```

在该程序中，首先初始化了字符串 `str`，并对串口进行了初始化，接着调用 `strstr` 函数返回字符串“line”处的指针，运行的结果将通过串口 0 输出。

12.2.15 搜索字符函数

搜索字符函数用于搜索字符出现的位置，其函数原型如下：

```
char*strchr(char *s1,char c);
```

其中，`s1` 为目标字符串，`c` 为待搜索的字符。该函数的功能是搜索字符串 `s1` 中是否包含字符 `c`，如果包含则返回第一次指向该字符的指针，否则返回 `NULL`。被搜索的字符可以是串结束符，此时返回值是指向串结束符的指针。程序示例如下：

```

#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    static char str[50]="this is line 1 ";
    char * p;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    p=strchr(str,'l');                             //调用 strchr 函数
    if(p!=NULL)

```

```
printf("found a 'l' at %s\n",p);           //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
found a 'l' at line 1
```

在该程序中首先初始化了字符串 str，并对串口进行了初始化，接着调用 strchr 函数查找字符“l”，并返回指向该字符的指针，运行的结果将通过串口 0 输出。

12.2.16 返回位置值的字符搜索函数

返回位置值的字符搜索函数用于搜索并返回字符出现的位置，其函数原型如下：

```
int strpos(char*s1,char c);
```

其中，s1 为目标字符串，c 为搜索的字符。该函数的功能是查找并返回字符 c 在字符串 s1 中第一次出现的位置值，没有找到该字符则返回-1，s1 串首字符的位置值是 0。strpos 函数的功能与 strchr 类似，只不过返回值不同。程序示例如下：

```
#include<string.h>           //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)               //主函数
{
static char str[50]="this is line 1 ";
int i;

SCON=0x50;                   //串口模式 1，允许接收
TMOD|=0x20;                   //初始化 T1 为定时功能，模式 2
PCON|=0x80;                   //设置 SMOD=1
TL1=0xF4;                     //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90;                     //中断
TR1=1;                        //启动定时器

i=strpos(str,'l');            //调用 strpos 函数
if(i!=-1)
printf("found a 'l' at %d\n",i); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
found a 'l' at 8
```

在该程序中，首先初始化了字符串 str，并对串口进行了初始化，接着调用 strpos 函数查找字符“l”，并返回位置值，运行的结果将通过串口 0 输出。

12.2.17 字符包含函数

字符包含函数用于检查字符串中是否包含某字符，其函数原型如下：

```
char*strrch(char *s1,char c);
```

其中，s1 为目标字符串，c 为查找的字符。该函数的功能是搜索字符串 s1 中是否包含字符 c，如果包含则返回最后一次指向该字符的指针，否则返回 NULL。被搜索的字符可以是串结束符，此时返回值是指向串结束符的指针。程序示例如下：

```
#include<string.h>           //头文件
#include<stdio.h>
#include <reg51.h>
```

```

void main(void)                                     //主函数
{
    static char str[50]="this is line 1 ";
    char * p;

    SCON=0x50;                                       //串口模式 1，允许接收
    TMOD|=0x20;                                     //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    p=strrchr(str, 's');                             //调用 strrchr 函数
    if(p!=NULL)
        printf("found the last 's' at %s\n", p);    //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
found the last 's' at s line 1
```

在该程序中，首先初始化了字符串 `str`，并对串口进行了初始化，接着调用 `strchr` 函数查找字符“s”，并返回指向该字符的指针，运行的结果将通过串口 0 输出。

12.2.18 返回位置值的字符包含函数

返回位置值的字符包含函数同样用于检查字符串中是否包含某字符，其函数原型如下：

```
int strrpos(char*s1, char c);
```

其中，`s1` 为目标字符串，`c` 为查找的字符。该函数的功能是查找并返回字符 `c` 在字符串 `s1` 中最后一次出现的位置值，没有找到该字符则返回 -1，`s1` 串首字符的位置值是 0。`strrpos` 函数的功能与 `strchr` 类似，只不过返回值不同。程序示例如下：

```

#include<string.h>                                   //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                                     //主函数
{
    static char str[50]="this is line 1 ";
    int i;

    SCON=0x50;                                       //串口模式 1，允许接收
    TMOD|=0x20;                                     //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    i=strrpos(str, 's');                             //调用 strrpos 函数
    if(i!=-1)
        printf("found the last 's' at %d\n", i);    //输出结果
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
found the last 's' at 6
```

在该程序中，首先初始化了字符串 `str`，并对串口进行了初始化，接着调用 `strstrpos` 函数查找字符“s”，并返回最后一次出现的位置值，运行的结果将通过串口 0 输出。

12.2.19 在指定字符集中查找不包含字符函数

在指定字符集中查找不包含字符函数用于查找不包含在指定字符集中的字符，其函数原型如下：

```
int strstrpn(char*s1,char*set);
```

其中，`s1` 为目标字符串，`set` 为字符集。该函数的功能是搜索字符串 `s1` 中第一个不包含在 `set` 串中的字符，返回值是字符串 `s1` 中包括在 `set` 中的字符的个数。如果 `s1` 中所有的字符都包含在 `set` 中，则返回 `s1` 的长度（不包括结束符）。如果 `set` 是空字符串则返回 0。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                                    //主函数
{
    static char str[50]="this is line 12";
    char set[]="this line 12345";
    int i;

    SCON=0x50;                                     //串口模式 1，允许接收
    TMOD|=0x20;                                     //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                         //启动定时器

    i=strstrpn(str,set);
    printf("strstrpn(str,set)=%d\n",i);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
strstrpn(str,set)=15
```

在该程序中，首先初始化了字符串 `str` 及 `set`，并对串口进行了初始化，接着调用 `strstrpn` 函数，并将运行的结果通过串口 0 输出。

12.2.20 在指定字符集中查找包含字符函数

在指定字符集中查找包含字符函数用于查找包含在指定字符集中的字符，其函数原型如下：

```
int strcspn(char*s1,char*set);
```

其中，`s1` 为目标字符串，`set` 为字符集。该函数的功能是搜索的是第一个包含在 `set` 串中字符，返回值是字符串 `s1` 中包括在 `set` 中的字符的个数。如果 `s1` 中所有的字符都包含在 `set` 中，则返回 `s1` 的长度（不包括结束符）。如果 `set` 是空字符串则返回 0。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
```

```

{
static char str[50]="this is line 12";
char set[]="12345";
int i;

    SCON=0x50;                //串口模式 1, 允许接收
    TMOD|=0x20;                //初始化 T1 为定时功能, 模式 2
    PCON|=0x80;                //设置 SMOD=1
    TL1=0xF4;                  //波特率 4800bit/s, 初值
    TH1=0xF4;
    IE|=0x90;                  //中断
    TR1=1;                     //启动定时器

i=strcspn(str,set);            //调用 strcspn 函数
printf("strcspn(str,set)=%d\n",i);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行, 运行的结果如下:

```
strcspn(str,set)=13
```

在该程序中, 首先初始化了字符串 `str` 及 `set`, 并对串口进行了初始化, 接着调用 `strcspn` 函数, 并将运行的结果通过串口 0 输出。

12.2.21 查找第一个包含字符函数

查找第一个包含字符函数用于查找第一个包含在指定字符集中的字符, 其函数原型如下:

```
char*strpbrk(char *s1,char*set);
```

其中, `s1` 为目标字符串, `set` 为字符集。该函数的功能是搜索字符串 `s1` 中第一个包含在 `set` 串中的字符, 返回值指向搜索到的字符的指针, 如果未找到, 则返回 `NULL`。程序示例如下:

```

#include<string.h>                //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                    //主函数
{
static char str[50]="this is line 12";
char set[]="12345";
char * p;

    SCON=0x50;                //串口模式 1, 允许接收
    TMOD|=0x20;                //初始化 T1 为定时功能, 模式 2
    PCON|=0x80;                //设置 SMOD=1
    TL1=0xF4;                  //波特率 4800bit/s, 初值
    TH1=0xF4;
    IE|=0x90;                  //中断
    TR1=1;                     //启动定时器

p=strpbrk(str,set);              //调用 strpbrk 函数
printf("p=strpbrk(str,set)=%s\n",p);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行, 运行的结果如下:

```
p=strpbrk(str,set)=12
```

在该程序中, 首先初始化了字符串 `str` 及 `set`, 并对串口进行了初始化, 接着调用 `strpbrk` 函

数，并将运行的结果通过串口 0 输出。

12.2.22 查找最后一个包含字符函数

查找最后一个包含字符函数用于查找最后一个包含在指定字符集中的字符，其函数原型如下：

```
char*strrpbrk(char *s1,char*set);
```

其中，s1 为目标字符串，set 为字符集。该函数的功能是搜索字符串 s1 中最后一个包含在 set 串中的字符，返回值指向搜索到的字符的指针，如果未找到，则返回 NULL。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    static char str[50]="this is line 12";
    char set[]="12345";
    char * p;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                      //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                      //中断
    TR1=1;                                         //启动定时器

    p=strrpbrk(str,set);                          //调用 strrpbrk 函数
    printf("p=strrpbrk(str,set)=%s\n",p);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
p=strrpbrk(str,set)=2
```

在该程序中，首先初始化了字符串 str 及 set，并对串口进行了初始化，接着调用 strrpbrk 函数，并将运行的结果通过串口 0 输出。

12.3 I/O 函数

I/O 函数主要用于数据的输入输出等操作，C51 语言中的 I/O 库函数的原型声明包含在头文件 STDIO.H 中。这些 I/O 函数使用 8051 单片机的串行接口进行通信，因此，在使用之前需要先进行串口的初始化。示例如下：

```
SCON=0x50;                                        //串口模式 1，允许接收
TMOD|=0x20;                                        //初始化 T1 为定时功能，模式 2
PCON|=0x80;                                        //设置 SMOD=1
TL1=0xF4;                                          //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90;                                         //中断
TR1=1;                                             //启动定时器
```

上面的语句采用 11.0592MHz 的时钟频率，分别设置了串口模式及波特率等，最后启动定时器来完成串口的初始化。下面分别介绍常用的 I/O 函数。

12.3.1 字符读入函数

字符读入函数用于从串口读入一个字符，其函数原型如下：

```
char _getkey(void);
```

该函数执行的操作是等待从 8051 的串口读入一个字符，并返回读入的原字符。程序示例如下：

```
#include<stdio.h>                                //头文件
#include <reg51.h>

void main(void)                                   //主函数
{
    char ch;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    while((ch=_getkey ())!=0x0d)                  //获取字符
    {
        printf("key=%c %bx\n",ch,ch);            //输出结果
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，当字符输入 e 时，运行的结果如下：

```
key=e 65
```

在该程序中，首先对串口进行了初始化，接着调用_getkey 函数获取字符，并通过串口 0 输出该字符及对应的十六进制数。当输入回车字符的时候退出程序。

12.3.2 字符读入输出函数

字符读入输出函数用于从串口读入一个字符并输出该字符，其函数原型如下：

```
char getchar (void);
```

该函数与_getkey 函数有细微的不同，其执行的操作是使用_getkey 从串口读入的一个字符，然后使用 putchar 函数将读入的字符输出。程序示例如下：

```
#include<stdio.h>                                //头文件
#include <reg51.h>

void main(void)                                   //主函数
{
    char ch;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
```

51 单片机开发与应用技术详解

```
TR1=1; //启动定时器

while((ch=getchar())!=0x0d) //获取字符
{
    printf("key=%c %bx\n",ch,ch); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，当字符输入 e 时，运行的结果如下：

```
ekey=e 65
```

在该程序中，首先对串口进行了初始化，接着调用 getchar 函数获取字符，并通过串口 0 输出该字符及对应的十六进制数。

12.3.3 字符串读入函数

字符串读入函数用于从串口读入一个字符串，其函数原型如下：

```
char *gets (char*s, int n);
```

其中，s 保存读入的字符串，n 为字符串的长度。该函数执行的操作是使用 getchar 函数从串口读入一个长度为 n 的字符串，并存入字符数组 s 中。如果遇到换行符，则结束字符的输入。输入成功时将返回传入的参数指针，失败时返回空指针 NULL。程序示例如下：

```
#include<stdio.h> //头文件
#include <reg51.h>

void main(void) //主函数
{
    xdata char str[50];

    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    gets(str, sizeof(str));
    printf("str=%s\n",str);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，当输入 hello everyone 时，运行的结果如下：

```
str=hello everyone
```

在该程序中，首先对串口进行了初始化，接着调用 gets 函数获取字符，并通过串口 0 输出该字符串。

12.3.4 字符回送函数

字符回送函数用于将输入的字符回送到输入缓冲区，其函数原型如下：

```
char ungetchar (char c);
```

其中，c 为输入字符。该函数执行的操作是将输入的字符回送到输入缓冲区，如果函数调用成功则返回 char 型值 c，失败时则返回 EOF。程序示例如下：

```
#include<stdio.h> //头文件
#include<ctype.h>
#include <reg51.h>
```

```

void main(void)                                //主函数
{
    char k;

    SCON=0x50;                                //串口模式 1，允许接收
    TMOD|=0x20;                                //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                //设置 SMOD=1
    TL1=0xF4;                                  //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                  //中断
    TR1=1;                                     //启动定时器

    while(isdigit(k=getchar ()))                //判断输入的是否为数字
    {}
    ungetchar(k);                              //调用 ungetchar 函数
}

```

该程序可以在 Keil μ Vision3 编译环境中执行。在该程序中，首先对串口进行了初始化，接着调用 `getchar` 函数及 `isdigit` 函数判断获取的字符是否为数字，如果是则一直循环接收，否则将输出该字符。

12.3.5 字符输出函数

字符输出函数用于通过 8051 串行口输出字符，其函数原型如下：

```
char putchar (char c);
```

其中，`c` 为通过 8051 串行口输出的字符。程序示例如下：

```

#include<stdio.h>                                //头文件
#include<ctype.h>
#include <reg51.h>

void main(void)                                //主函数
{
    char k;

    SCON=0x50;                                //串口模式 1，允许接收
    TMOD|=0x20;                                //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                //设置 SMOD=1
    TL1=0xF4;                                  //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                  //中断
    TR1=1;                                     //启动定时器

    for(k=0x30; k<=0x3A; k++)                //循环输出字符
        putchar(k);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
0123456789
```

在该程序中，首先对串口进行了初始化，接着调用 `putchar` 函数通过串口 0 输出 10 个字符。

12.3.6 格式化输出函数

格式化输出函数用于按照一定的格式输出数据或字符串，其函数原型如下：

```
int printf (const char *fmstr[,argument]...);
```

该函数的功能是以一定的格式通过 8051 单片机的串行口输出数值和字符串。其中，第一个参数 fmstr 是格式控制字符串，参数 argument 可以是字符串指针、字符或数值，该函数的返回值为实际输出的字符个数。

这里需要注意的是，允许作为 printf 参数的总字节数受 8051 单片机结构上的存储空间限制。在 Keil μ Vision3 编译环境下，如果采用 SMALL 和 COMPACT 编译模式，可传递最多 15 个字节的参数（即 5 个指针，或者 1 个指针和 3 个长字），而在 LARGE 编译模式下，则可传递最多 40 个字节的参数。

另外，这里的格式控制字符串的形式如下所示，方括号内是可选项。

```
%[flag][width][.percision][{b[B][l]L}]type
```

其中，flag 为标志字符，用于控制输出位置、符号、小数点及八进制的十六进制的前缀等。在 C51 语言中支持的 flag 标志字符，如表 12-1 所示。

表 12-1 flag 标志字符

flag 标志字符	意 义
-	左对齐输出
+	如果是输出有符号数值，则在前面加上+/-号
空格	如果输出为正值，则在左边补空格，否则不显示空格
#	当与字符 0、x 或 X 一起使用时，在非 0 输出值前面加上 0、0x、0X；当与字符 g、o、f、e 或 EX 一起使用时，在输出值中包含一个十进制的小数点；在其他情况下将忽略该标志

width 是一个正的十进制数，其用来定义显示的字符数。如果实际输出显示的字符数小于 width 值，则在其左边以空格补齐，如果 width 值以非 0 开始，则需要在其左边补 0。

percision 由小圆点“.”加上一个非负的十进制整数构成，其用来表示输出精度。使用 percision 指定精度时，将使输出的浮点数进行四舍五入。在 C51 语言中，可以设置输出精度来控制输出字符的数目、整数值的位数或浮点数的有效位数等。不同的输出格式，精度具有不同的含义，使用时需要注意。

可选字符 b 或 B 和 l 或 L 用与和格式转换字符一起使用，其意义如表 12-2 所示。

表 12-2 可选字符 b、B、l、L 的意义

可选字符	意 义
b、B	与格式类型字符 d、o、u、x 或 X 一起使用，将接受[unsighed]char 参数类型，如%bu、%bx 等
l、L	与格式类型字符 d、o、u、x 或 X 一起使用，将接受[unsighed]long 参数类型，如%ld、%lx 等

type 为输出格式转换字符，其意义如表 12-3 所示。

表 12-3 type 选项及其意义

格式转换字符	类 型	输出格式
c	char	字符
d	int	有符号十进制数（16 位）
u	unsighed int	无符号十进制数
o	unsighed int	无符号八进制数
x,X	unsighed int	无符号十六进制数
f	float	(-)dddd.dddd 形式的浮点数
e,E	float	(-)d.ddddE[sign]dd 形式的浮点数
g,G	float	输出 e 或 f 形式中更紧凑的一种
s	指针	字符串
p	指针	带存储器类型标志和偏移的指针 M:aaaa

其中 M:=C(ode), D(ata), l(data), P(data); a:=指针偏移量

下面举例讲解 printf 函数的使用，程序示例如下：

```
#include<string.h> //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void) //主函数
{
char a = 1; //变量声明并赋初值
int b = 12345;
long c = 0x7FFFFFFF;

unsigned char x = 'A';
unsigned int y = 54321;
unsigned long z = 0x5A6F7E00;

float f = 21.0; //浮点型
float g = 13.64;

char buf [] = "This is test string"; //字符串
char *p = buf;

SCON=0x50; //串口模式 1，允许接收
TMOD|=0x20; //初始化 T1 为定时功能，模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器

printf ("char %d int %d long %ld\n",a,b,c); //格式化输出
printf ("Uchar %bu Uint %u Ulong %lu\n",x,y,z); //格式化输出
printf ("xchar %bx xint %x xlong %lx\n",x,y,z); //格式化输出
printf ("String %s is at address %p\n",buf,p); //格式化输出
printf ("%f != %g\n", f, g); //格式化输出
printf ("%*f != %*g\n", (int)8, f, (int)8, g); //格式化输出
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
char 1 int 12345 long 2147483647
Uchar 65 Uint 54321 Ulong 1517256192
xchar 41 xint d431 xlong 5a6f7e00
String This is test string is at address I:0038
21.000000 !=13.64
21.000000 != 13.64
```

在该程序中，首先声明并初始化一些变量，然后对串口进行了初始化，接着调用 printf 函数进行格式化输出，结果通过串口 0 输出。读者可以根据前面介绍的格式化知识分析其输出格式。

12.3.7 格式化内存缓冲区输出函数

格式化内存缓冲区输出函数用于按照一定的格式将数据或字符串输出到内存缓冲区中，其

函数原型如下：

```
int sprintf (char*s,const char *fmstr[,argument]...);
```

该函数执行的操作是通过指针 s，将字符串送入内存数据缓冲区，并以 ASCII 码的形式储存。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
char buf [50];                                    //声明变量
int n;
int a,b;
float pi;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                      //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                      //中断
    TR1=1;                                         //启动定时器

a = 123;                                          //变量赋值
b = 456;
pi = 3.14159;

n = sprintf (buf, "%f\n", 2.7);                  //调用 sprintf 函数
n+= sprintf (buf+n, "%d\n", a);
n+= sprintf (buf+n, "%d %s %g", b, "---", pi);
puts (buf);                                     //输出
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
2.700000
123
456 --- 3.14159
```

在该程序中，首先声明变量，然后对串口及变量进行了初始化，接着调用 sprintf 函数格式化字符串并保存在 buf 中，最后使用 puts 函数将结果通过串口 0 输出。

12.3.8 字符串输出函数

字符串输出函数用于将字符串和换行符写入串行口，其函数原型如下：

```
int puts (const char*s);
```

其中，s 为输出的字符串或换行符。如果执行成功则返回 0，错误时返回 EOF。程序示例如下：

```
#include<stdio.h>                                //头文件
#include <reg51.h>

voidmain(void)                                    //主函数
{
    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
```

```
TL1=0xF4; //波特率 4800bit/s , 初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器
puts("This is a test line"); //输出字符串
puts("This is another test line");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
This is a test line
This is another test line
```

在该程序中，首先初始化串口，接着调用 puts 函数通过串口 0 输出字符串。

12.3.9 格式化输入函数

格式化输入函数用于将字符串和数据按照一定的格式从串口读入，其函数原型如下：

```
int scanf (const char *fmstr[,argument]...);
```

该函数的功能是在格式字符的控制下从串行口读入数据。其中每个参数都必须是指针。

scanf 返回值是所发现并转换的输入项数，如遇到错误则返回 EOF。

该函数的格式控制字符串形式如下所示，方括号内是可选项。

```
%[*][width] [{B[h][l]}]type
```

其中，width 用来控制输入数据的最大长度或字符数目，其是十进制正整数。该函数从输入数据流中读出不超过规定宽度的字符，并转换到相应的变量中。这里需要注意，如果先输入一个空格符或无法识别的字符，这样读入的字符数将会小于需要的宽度值。可选字符 B、h、l 与输入格式转换字符一起使用，其含义如表 12-4 所示。

表 12-4 可选字符 B、h、l 的意义

可选字符	意 义
B、h	用于格式类型字符 d、o、u、x 的前缀，可将参数定义为字符指针，指示输入整型数，如%bu、%bx 等
l	用于格式类型字符 d、o、u、x 的前缀，可将参数定义为长指针，指示输入长整数，如%lu、%lx 等

type 是输入格式转换字符，其含义如表 12-5 所示。

表 12-5 type 选项及其意义

格式转换字符	类 型	输出格式
C	char*	字符
S	char*	字符串
D	int*	有符号十进制数（16 位）
I	int*	有符号十进制、八进制数、十六进制
U	unsighed int*	无符号十进制数
O	unsighed int*	无符号八进制数
X	unsighed int*	无符号十六进制数
f,e,g	float*	浮点数

使用本函数的示例如下：

```
#include<string.h> //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void) //主函数
{
char a;
int b;
```


51 单片机开发与应用技术详解

```
long c;
unsigned char x;
unsigned int y;
unsigned long z;
float f,g;
char d, buf [10];
int argsread;

    SCON=0x50;                                //串口模式 1，允许接收
    TMOD|=0x20;                                //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                //设置 SMOD=1
    TL1=0xF4;                                  //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                  //中断
    TR1=1;                                     //启动定时器

printf ("Enter a signed byte, int, and long\n");
argsread = scanf ("%bd %d %ld", &a, &b, &c);    //输入整型数据
printf ("%d arguments read.a=%c,b=%d,c=%ld\n", argsread,a,b,c);
printf ("Enter an unsigned byte, int, and long\n");

argsread = scanf ("%bu %u %lu", &x, &y, &z);    //输入无符号型数据
printf ("%d arguments read.x=%c,y=%u,z=%lu\n", argsread,x,y,z);
printf ("Enter a character and a string\n");

argsread = scanf ("%c", &d);                    //输入字符
printf ("%d arguments read.d=%c\n", argsread,d);
printf ("Enter a string\n");

argsread = scanf ("%9s",buf);                    //输入字符串
printf ("%d arguments read.buf=%s\n", argsread,buf);
printf ("Enter two floating-point numbers\n");

argsread = scanf ("%f %f", &f, &g);            //输入浮点型数据
printf ("%d arguments read.f=%f,g=%f\n", argsread,f,g);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中，首先初始化串口，然后分别调用 scanf 函数从串口 0 获得输入的数据，然后使用 printf 函数输出。

12.3.10 格式化内存缓冲区输入函数

格式化内存缓冲区输入函数用于将格式化的字符串和数据送入数据缓冲区，其函数原型如下：

```
int sscanf (char*s,const char *fmstr[,argument]...);
```

该函数的功能是将输入的字符串通过指针 s 指向的数据缓冲区。输入数据根据格式控制字符串 fmstr 被存放到由 argument 指定的地址。其他方面 sscanf 函数与 scanf 函数类似。程序示例如下：

```
#include<string.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    char a;
    int b;
    long c;
    unsigned char x;
    unsigned int y;
    unsigned long z;
    float f,g;
    char d, buf [10];
```

```

int argsread;

SCON=0x50;                //串口模式 1，允许接收
TMOD|=0x20;               //初始化 T1 为定时功能，模式 2
PCON|=0x80;               //设置 SMOD=1
TL1=0xF4;                 //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90;                 //中断
TR1=1;                   //启动定时器

printf ("Reading a signed byte, int,and long\n");
argsread = sscanf ("2 -123 456789", "%bd %d %ld", &a, &b, &c);
printf ("%d arguments read.a=%bd,b=%d,c=%ld\n", argsread,a,b,c);
printf ("Reading an unsigned byte, int, and long\n");
argsread = sscanf ("1 56 87654321", "%bu %u %lu", &x, &y, &z);
printf ("%d arguments read.x=%bu,y=%u,z=%lu\n", argsread,x,y,z);
printf ("Reading a character and a string\n");
argsread = sscanf ("A ABCDEFG", "%c %9s", &d, buf);
printf ("%d arguments read.d=%c,buf=%s\n", argsread,d,buf);
printf ("Reading two floating-point numbers\n");
argsread = sscanf ("23.5 12.7", "%f %f", &f, &g);
printf ("%d arguments read.f=%f,g=%f\n", argsread,f,g);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

Reading a signed byte, int,and long
3 arguments read.a=2,b=-123,c=456789
Reading an unsigned byte, int,and long
3 arguments read.x=1,y=56,z=87654321
Reading a character and a string
2 arguments read.d=A,buf=ABCDEFG
Reading two floating-point numbers
2 arguments read.f=23.500000,g=12.700000

```

在该程序中，首先初始化串口，接着调用 `sscanf` 函数获取字符串及数据，最后通过串口 0 输出结果。

12.3.11 字符串内存输出函数

字符串内存输出函数用于将格式化字符串输出到内存数据缓冲区，其函数原型如下：

```
int vprintf (const char *fmstr,char *argptr);
```

其中，`fmstr` 为格式化字符串，`argptr` 指向变量表的指针而不是变量表，函数返回值为实际写入到输出字符串中的字符数。其他方面 `vprintf` 函数与 `printf` 函数类似。程序示例如下：

```

#include <stdarg.h>                //头文件
#include<stdio.h>
#include <reg51.h>
void error (char *fmt, ...)        //定义 error 函数
{
    va_list arg_ptr;
    va_start (arg_ptr, fmt);        //格式化字符串
    vprintf (fmt, arg_ptr);          //调用 Vprintf 函数
    va_end (arg_ptr);
}

void main(void)                    //主函数
{
    int i;
    i = 1000;
}

```

```
SCON=0x50; //串口模式 1，允许接收
TMOD|=0x20; //初始化 T1 为定时功能，模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器
error ("Error: '%d' number too large\n", i); //调用 error 函数
error ("This is a syntax Error\n");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Error: '1000' number too large
This is a syntax Error
```

在该程序中，首先定义了错误处理函数 error，该函数中调用了 vprintf 函数。在主程序中，首先初始化串口，接着调用自定义函数 error 通过串口 0 输出错误信息。

12.3.12 指向缓冲区的输出函数

指向缓冲区的输出函数用于将格式化字符串和数字输出到由指针所指向的内存数据缓冲区，其函数原型如下：

```
int vsprintf (char*s,const char *fmstr,char *argptr);
```

该函数执行的操作是将格式化字符串和数字输出到由指针所指向的内存数据缓冲区。其中，该函数接受的是一个指向变量表的指针而不是变量表，其返回值为实际写入到输出字符串中的字符数。其他方面 vsprintf 函数与 sprintf 函数类似。程序示例如下：

```
#include <stdarg.h> //头文件
#include<stdio.h>
#include <reg51.h>
xdata char etxt[30]; //字符串缓冲区
void error (char *fmt, ...) //自定义函数
{
    va_list arg_ptr;
    va_start (arg_ptr, fmt); //格式化字符串
    vsprintf (etxt, fmt, arg_ptr); //调用 vsprintf 函数
    va_end (arg_ptr);
}
void main(void) //主函数
{
    int i;
    i = 1000;
    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器
    error ("Error: '%d' number too large\n", i); //调用 error 函数
    error ("This is a syntax Error\n");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Error: '1000' number too large
This is a syntax Error
```

在该程序中，首先定义了错误处理函数 `error`，该函数中调用了 `vsprintf` 函数。在主程序中，首先初始化串口，接着调用自定义函数 `error` 通过串口 0 输出错误信息。

12.4 数学函数

数学函数主要用于进行数学运算，其原型声明包含在头文件 `MATH.H` 中。下面介绍一些常用的数学函数。

12.4.1 绝对值函数

绝对值函数用于计算并返回输出数据的绝对值。按照其操作数的数据类型不同，有如下几种形式。

```
int abs(int val);
char cabs(char val);
float fabs(float val);
long labs(long val)
```

这些函数分别用于计算整型、字符型、浮点型及长整型数据的绝对值。程序示例如下：

```
#include<math.h> //头文件
#include<stdio.h>
#include <reg51.h>
void main(void) //主函数
{
    int a=-12; //声明并初始化变量
    char b=-43;
    float c=-4.56;
    long d=-123456789;

    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    printf("abs(%d)=%d\n",a,abs(a)); //调用 abs 函数
    printf("cabs(%d)=%d\n",b,cabs(b)); //调用 cabs 函数
    printf("fabs(%f)=%f\n",c,fabs(c)); //调用 fabs 函数
    printf("labs(%ld)=%ld\n",d,labs(d)); //调用 labs 函数
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
abs(-12)=12
cabs(-43)=43
fabs(-4.560000)=4.560000
labs(-123456789)=123456789
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，最后分别调用 `abs`、`cabs`、`fabs` 和 `labs` 函数求绝对值并通过串口 0 输出。

12.4.2 指数及对数函数

指数函数用于计算并返回输出数据的指数。对数函数用于计算并返回输出数据的对数。其函数原型如下：

```
float exp(float x);
float log(float x);
float log10(float x);
float sqrt(float x);
```

其中，exp 函数用于计算并返回浮点数 x 的指数，log 函数用于计算并返回浮点数 x 的自然对数（自然对数以 e 为底，e=2.718282），log10 函数用于计算并返回浮点数 x 的以 10 为底的对数值，sqrt 函数用于计算并返回浮点数 x 的平方根。程序示例如下：

```
#include<math.h> //头文件
#include<stdio.h>
#include <reg51.h>

void main(void) //主函数
{
    float x=1.5; //声明并初始化变量
    float y=2.718282;
    float z=100;

    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    printf("exp(%f)=%f\n",x,exp(x)); //调用 exp 函数
    printf("log(%f)=%f\n",y,log(y)); //调用 log 函数
    printf("log10(%f)=%f\n",z,log10(z)); //调用 log10 函数
    printf("sqrt(%f)=%f\n",z,sqrt(z)); //调用 sqrt 函数
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
exp(1.500000)=4.481689
log(2.718282)=1.000000
log10(100.000000)=2.000000
sqrt(100.000000)=10.000000
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，最后分别调用 exp、log、log10 和 aqrt 函数求值并通过串口 0 输出。

12.4.3 三角函数

三角函数用于计算数学中三角函数的值。在 C51 语言中包含如下几种三角函数。

```
float cos(float x);
float sin(float x);
float tan(float x);
float acos(float x);
float asin(float x);
float atan(float x);

float atan2(float y, float x);
float cosh(float x);
float sinh(float x);
float tanh(float x);
```

其中, `cos` 函数用于计算并返回余弦值, `sin` 函数用于计算并返回正弦值, `tan` 函数用于计算并返回正切值。所有函数的变量范围都是 $-\pi/2 \sim +\pi/2$, 变量值必须在 ± 65535 之间, 否则产生一个 NaN 错误。

`acos` 函数用于计算并返回反余弦值, `asin` 函数用于计算并返回反正弦值, `atan` 函数用于计算并返回反正切值, 它们的值域为 $-\pi/2 \sim +\pi/2$ 。

`atan2` 函数用于计算并返回 y/x 的反正切值。`cosh` 函数用于计算并返回双曲余弦值, `sinh` 函数用于计算并返回双曲正弦值, `tanh` 函数用于计算并返回双曲正切值。下面分别介绍其在程序中的用法, 程序示例如下:

```
#include<math.h>           //头文件
#include<stdio.h>
#include <reg51.h>

const double PI=3.1415926;

voidmain(void)             //主函数
{
    float x=PI/4;           //声明并初始化变量
    float y=0.5;
    float z=2;

    SCON=0x50;              //串口模式 1, 允许接收
    TMOD|=0x20;             //初始化 T1 为定时功能, 模式 2
    PCON|=0x80;             //设置 SMOD=1
    TL1=0xF4;               //波特率 4800bit/s, 初值
    TH1=0xF4;
    IE|=0x90;               //中断
    TR1=1;                  //启动定时器

    printf("cos(%f)=%f\n",x,cos(x)); //调用 cos 函数
    printf("sin(%f)=%f\n",x,sin(x)); //调用 sin 函数
    printf("tan(%f)=%f\n",x,tan(x)); //调用 tan 函数
    printf("acos(%f)=%f\n",y,acos(y)); //调用 acos 函数
    printf("asin(%f)=%f\n",y,asin(y)); //调用 asin 函数
    printf("atan(%f)=%f\n",y,atan(y)); //调用 atan 函数
    printf("atan2(%f,%f)=%f\n",y,z,atan2(y,z)); //调用 atan2 函数
    printf("cosh(%f)=%f\n",x,cosh(x)); //调用 cosh 函数
    printf("sinh(%f)=%f\n",x,sinh(x)); //调用 sinh 函数
    printf("tanh(%f)=%f\n",x,tanh(x)); //调用 tanh 函数
}
```

该程序可以在 Keil μ Vision3 编译环境中执行, 运行的结果如下:

```
cos(0.785398)=0.707107
sin(0.785398)=0.707107
tan(0.785398)=1
acos(0.500000)=1.047198
asin(0.500000)=0.523599
atan(0.500000)=0.463648
atan2(0.500000,2.000000)=0.244979
cosh(0.785398)=1.324609
sinh(0.785398)=0.868671
tanh(0.785398)=0.655794
```

在该程序中, 首先定义并初始化变量, 接着对串口进行初始化, 最后分别调用 `cos`、`sin`、

51 单片机开发与应用技术详解

tan、acos、asin、atan、atan2、cosh、sinh 和 tanh 函数求值并通过串口 0 输出。

12.4.4 取整函数

取整函数用于取输入数据的整数。在 C51 语言中，包含两类取整函数，原型如下：

```
float ceil(float x);
float floor(float x);
```

其中，ceil 函数用于计算并返回一个不小于 x 的最小正整数（作为浮点数），floor 函数用于计算并返回一个不大于 x 的最小正整数（作为浮点数）。程序示例如下：

```
#include<math.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                                  //主函数
{
    float x=4.5;                                  //声明并初始化变量

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    printf("ceil(%f)=%f\n",x,ceil(x));              //调用 ceil 函数
    printf("floor(%f)=%f\n",x,floor(x));            //调用 floor 函数
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
ceil(4.500000)=5.000000
floor(4.500000)=4.000000
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，最后分别调用 ceil 和 floor 函数求值并通过串口 0 输出。

12.4.5 浮点型分离函数

浮点型分离函数用于将浮点型数据的整数和小数部分分开，其函数原型如下：

```
float modf(float x,float*ip);
```

函数 modf 将浮点数 x 分成整数和小数两部分，整数部分放入*ip，返回值为小数部分。两者都含有与 x 相同的符号。程序示例如下：

```
#include<math.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                  //主函数
{
    float x=4.5,y,z;                              //声明变量

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
```

```
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器

y=modf(x, &z); //调用 modf 函数
printf("%f=%f+%f\n", x, y, z); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
4.500000=0.500000+4.000000
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，最后调用 modf 函数求解该浮点型数据的整数和小数部分，并通过串口 0 输出。

12.4.6 幂函数

幂函数用于进行幂指数运算。其函数原型如下：

```
float pow(float x,floaty);
```

该函数用于计算并返回 x^y 的值。如果 x 不等于 0 而 y=0，则返回 1。当 x=0 且 y<=0 或 x<0 且 y 不是整数时，返回 NaN。程序示例如下：

```
#include<math.h> //头文件
#include<stdio.h>
#include <reg51.h>

void main(void) //主函数
{
float x=2.0; //声明并初始化变量
float y=4.0 ;

SCON=0x50; //串口模式 1，允许接收
TMOD|=0x20; //初始化 T1 为定时功能，模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器

printf("%f^%f=%f\n",x,y,pow(x,y)); //调用 pow 函数
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
2.000000^4.000000=16.000000
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，最后调用 pow 函数求解幂运算，并通过串口 0 输出。

12.5 标准函数

标准函数主要用于完成数据类型转换及存储器分配等操作。标准函数的原型声明包含在头文件 STDLIB.H 中。下面介绍常用的一些函数。

12.5.1 字符串转换函数

字符串转换函数用于将字符串转换成数值类型并输出。根据输出数值类型的不同，可以有如下几种形式。

```
float atoi(char*sl);
float atol(char*sl);
float atof(char*sl);
```

其中，atoi 函数用于将字符串 sl 转换成整型数值并返回该值。输入字符串的格式为：[whitespace][{+/-}]数字。其中，whitespace 可由空格、/、制表符组成。这里的数字可以是一个或多个十进制数。

函数 atol 用于将字符串 sl 转成长整型数值并返回该值。输入字符串中必须包含与规定长整型数值相符的数。输入字符串的格式为：[whitespace][{+/-}]数字。其中，whitespace 可由空格、/、制表符组成。这里的数字可以是一个或多个十进制数。当该函数遇到第一个不能构成数字的字符时，便停止对输入字符的读取。

函数 atof 用于将字符串 sl 转换成浮点数值并返回该值，输入字符串中必须包含与浮点值规定相符的数。输入字符串的格式为：[{+/-}]数字[.数字][{e/E}[{+/-}]数字]。当该函数遇到第一个不能构成数字的字符时，便停止对输入字符的读取。

示例如下：

```
#include<stdlib.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                     //主函数
{
char str1[]="1234";                                //声明并初始化变量
char str2[]="987654321";
char str3[]="12.34";
int i;
long l;
float f;

SCON=0x50;                                         //串口模式 1，允许接收
TMOD|=0x20;                                       //初始化 T1 为定时功能，模式 2
PCON|=0x80;                                       //设置 SMOD=1
TL1=0xF4;                                         //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90;                                         //中断
TR1=1;                                           //启动定时器

i=atoi(str1);                                   //调用 atoi 函数
l=atol(str2);                                    //调用 atol 函数
f=atof(str3);                                    //调用 atof 函数

printf("atoi(%s)=%d\n",str1,i);                //输出结果
printf("atol(%s)=%ld\n",str2,l);
printf("atof(%s)=%f\n",str3,f);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
atoi(1234)=1234
atoll(987654321)=987654321
atof(12.34)=12.340000
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，最后分别调用 `atoi`、`atol` 和 `atof` 函数进行字符串转换，并通过串口 0 输出结果。

12.5.2 带返回指针的字符串转换函数

带返回指针的字符串转换函数将字符串转换成数值类型并输出，同时返回未转换部分的指针。根据输出数值类型的不同，可以有如下几种形式。

```
float strtod(const char*s,char**ptr);
long strtol (const char*s,char**ptr,unsigned char base);
unsigener long strtoul (const char*s,char**ptr,unsigned char base);
```

其中，函数 `strtod` 将字符串 `s` 转换成浮点型数据并返回该值。输入字符串的格式为：`[{+/-}]数字[.数字][{e/E}[{+/-}]数字]`。在进行转换的时候，字符串前面的空格、/、制表符将被忽略。函数 `strtol` 将字符串 `s` 转换成 `long` 型数值并返回该值。输入字符串的格式为：`[whitespace][{+/-}]数字`。其中，`whitespace` 可由空格、/、制表符组成，在进行转换时将被忽略。函数 `strtoul` 将字符串 `s` 转换成 `unsigned long` 型数值并返回该值，溢出时则返回 `ULONG_MAX`。输入字符串的格式为：`[whitespace][{+/-}]数字`。其中，`whitespace` 可由空格、/、制表符组成，在进行转换时将被忽略。

对于 `strtol` 函数和 `strtoul` 函数，`base` 值用于指定基数，通常具有如下意义。

- 如果 `base` 的值为 0，则数据为 10 进制、8 进制、16 进制格式。
- 如果 `base` 的值在 2~36 之间，表示制定基数的一个整数，字母 `a~z` 或 `A~Z` 表示数据 10~36，只有小于 `base` 的字母表示的值才有效。此时数据为一个字母或数字的非零序列。

在这几个函数中，`ptr` 指向字符串 `s` 中已经转换部分后面的第一个字符，如果不进行转换，则 `ptr` 被设为字符串的值，同时返回 0。这里举例讲解这些函数在程序中的应用。程序示例如下：

```
#include<stdlib.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
    char str1[]="12.34 Hello";                    //声明并初始化变量
    char str2[]="987654321 Hello";
    char str3[]="1234ABCD Hello";
    char * ptr;
    float fl;
    long l;
    unsigned long ul;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    fl=strtod(str1,&ptr);                          //调用 strtod 函数
    printf("strtod(%)=%f,%s\n",str1,fl,ptr);        //输出结果
    l=strtol(str2,&ptr,10);                         //调用 strtol 函数
    printf("strtol(%)=%ld,%s\n",str2,l,ptr);        //输出结果
```

51 单片机开发与应用技术详解

```
ul=strtoul(str3,&ptr,16); //调用 strtoul 函数
printf("strtoul(%s)=%lx,%s\n",str3,ul,ptr); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
strtod(12.34 Hello)=12.340000,Hello
strtol(987654321 Hello)=987654321, Hello
strtoul(1234ABCD Hello)=1234abcd, Hello
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，最后分别调用 strtod、strtol 和 strtoul 函数进行字符串转换，并通过串口 0 输出结果。

12.5.3 随机函数

随机函数用于产生伪随机数。在 C51 语言中，包含两种随机函数，其函数声明如下：

```
int rand();
void srand(int n);
```

其中，函数 rand 用于返回一个 0 到 32767 之间的伪随机数，而函数 srand 用来初始化随机数发生器的随机种子。如果不使用 srand 函数，则对 rand 函数的相继调用将产生相同的随机序列。程序示例如下：

```
#include<stdlib.h> //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void) //主函数
{
    int i; //声明并初始化变量

    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    srand(56); //随机种子
    for(i=0;i<10;i++)
    {
        printf("%d ",rand()); //输出随机数
    }
    printf("\n");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
25550 6948 10313 5703 13365 28382 17382 14853 19675 10064
```

在该程序中，首先定义并初始化变量，接着对串口进行初始化，然后调用 srand 函数初始化随机种子，接着调用 rand 函数通过串口 0 数出 10 个随机数。

12.5.4 数组内存分配函数

数组内存分配函数用于为 n 个元素的数组分配内存空间，其函数原型如下：

```
void*calloc (unsigned int n,unsigned int size);
```

其中，n 数组元素的个数，size 为数组中每个元素的大小。该函数所分配的内存区域用 0

进行初始化。返回值为已分配的内存单元起始地址，如果不成功则返回 0。程序示例如下：

```
#include<stdlib.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                                    //主函数
{
    int *p;                                         //指针指向有 20 个整型元素的数组

    SCON=0x50;                                     //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    p=calloc(20,sizeof(int));                      //调用 calloc 函数
    if(p==NULL)
        printf("Error allocating array\n");
    else
        printf("Array address is %p\n", (void*)p); //输出数组地址
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Array address is x:0000
```

在该程序中，首先定义变量，接着对串口进行初始化，然后调用 calloc 函数申请 20 个内存分配。如果申请成功，则通过串口 0 输出数组的地址，否则输出错误信息。

12.5.5 释放内存函数

释放内存函数用于释放前面已分配的内存空间，其函数原型如下：

```
void free (void xdata *p);
```

其中，指针 p 指向待释放的存储区域。p 必须是以前用 calloc、malloc 或 realloc 函数分配的存储区域，如果 p 为 NULL，则该函数无效。经 free 函数所被释放的存储区域可以参与以后的分配。程序示例如下：

```
#include<stdlib.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                                    //主函数
{
    int *p;                                         //指针指向有 20 个整型元素的数组

    SCON=0x50;                                     //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                     //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器
```

```
p=calloc(20,sizeof(int)); //申请内存
if(p==NULL)
    printf("Error allocating array\n");
else
{
    printf("Array address is %p\n", (void*)p); //输出地址
    free(p); //释放内存
    printf("Memory free\n");
}
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Array address is x:0000
Memory free
```

在该程序中，首先定义变量，接着对串口进行初始化，然后调用 calloc 函数申请 20 个内存分配。如果申请成功，则通过串口 0 输出数组的地址并通过 free 函数释放该内存，否则输出错误信息。

12.5.6 初始化内存函数

初始化内存函数用于对前面申请的内存进行初始化，其函数原型如下：

```
void init_mempool (void xdata *p,unsigned int size);
```

其中，指针 p 表示存储区首地址，size 表示存储区大小。该函数可对被函数 calloc、malloc、free 或 realloc 管理的存储区域进行初始化。程序示例如下：

```
#include<stdlib.h> //头文件
#include<stdio.h>
#include <reg51.h>

unsigned char xdata malloc_mempool [0x1000];

void main(void) //主函数
{
    int i; //声明变量
    xdata void *p;

    SCON=0x50; //串口模式 1，允许接收
    TMOD|=0x20; //初始化 T1 为定时功能，模式 2
    PCON|=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90; //中断
    TR1=1; //启动定时器

    //调用 init_mempool 函数
    init_mempool (&malloc_mempool, sizeof(malloc_mempool))

    p = malloc (100); //调用 malloc 函数

    for (i = 0; i < 100; i++) //循环赋值
        ((char *) p)[i] = i;

    free (p); //释放内存
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在该程序中，首先定义变量，接着对串口进行初始化，然后调用 `init_mempool` 函数和 `malloc` 函数初始化内存并申请内存空间。程序中通过循环赋值对申请的内存进行写操作，最后通过 `free` 函数释放内存。

12.5.7 内存分配函数

内存分配函数用于在内存中分配指定大小的存储空间，其函数原型如下：

```
void*malloc (unsigned int size);
```

其中，`size` 为分配的空间大小，返回值为指向所分配内存的指针。如果返回 `NULL`，则表示没有足够的内存空间可用。程序示例如下：

```
#include<stdlib.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)                                    //主函数
{
    int *p;                                         //指针指向整型元素

    SCON=0x50;                                     //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                         //启动定时器

    p=malloc(20);                                  //申请内存
    if(p==NULL)
        printf("Error allocating array\n");
    else
        printf("Array address is %p\n", (void*)p); //输出地址
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Array address is x:0000
```

在该程序中，首先定义变量，接着对串口进行初始化，然后调用 `malloc` 函数申请 20 个内存分配。如果申请成功，则通过串口 0 输出地址，否则输出错误信息。

12.5.8 调整内存大小函数

调整内存大小函数用于调整先前分配的存储器区域大小，其函数原型如下：

```
void*realloc (void xdata *p,unsigned int size);
```

其中，参数 `p` 表示该存储区域的起始地址，参数 `size` 表示新分配的存储区域大小。该函数的返回值为新区域所指向的指针。如果返回 `NULL`，则表示没有足够的内存空间可用。原存储器区域的内容被复制到新存储器区域中，如果新存储器区域较大，多出的区域不作初始化。程序示例如下：

```
#include<stdlib.h>                                //头文件
#include<stdio.h>
#include <reg51.h>

voidmain(void)                                    //主函数
{
```

51 单片机开发与应用技术详解

```
int *p; //指针指向整型元素
int *newp;

SCON=0x50; //串口模式 1，允许接收
TMOD|=0x20; //初始化 T1 为定时功能，模式 2
PCON|=0x80; //设置 SMOD=1
TL1=0xF4; //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90; //中断
TR1=1; //启动定时器

p=malloc(20); //申请内存
if(p==NULL)
    printf("Error allocating array\n");
else
{
    printf("Array address is %p\n", (void*)p); //输出地址
    newp= realloc(p,200); //重新分配内存
    if(newp==NULL)
        printf("Error allocating array\n");
    else
        printf("Array address is %p\n", (void*)newp); //输出地址
}
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Array address is x:0000
Array address is x:0000
```

在该程序中，首先定义变量，接着对串口进行初始化，然后调用 malloc 函数申请 20 个内存分配。如果申请成功，则通过串口 0 输出地址，接着使用 realloc 函数重新分配 200 个内存空间。最后，通过串口 0 输出相关信息。

12.6 内部函数

内部函数的原型声明包含在头文件 INTRINS.H 中。下面介绍常用的一些内部函数。

12.6.1 循环左移函数

循环左移函数主要用于将数据按照二进制循环左移 n 位。按照操作数据类型的不同，其函数原型如下几种形式。

```
unsigned char _crol_(unsigned char val,unsigned char n);
unsigned int _lrol_(unsigned int val,unsigned char n);
unsigned long _lrol_(unsigned long val,unsigned char n);
```

其中，val 为待移位的变量，n 为循环移位的次数。函数_crol_、_lrol_和_lrol_分别用于字符型、整型和长整型变量的循环左移，其返回值分别为移位后的字符型、整型和长整型。该函数与 8051 单片机的 RLA 指令相关。程序示例如下：

```
#include <intrins.h> //头文件
#include <stdio.h>
#include <reg51.h>

void main(void) //主函数
{
    char a=0xA5; //声明并初始化变量
```

```

int b=13;
long c=123456789;

SCON=0x50;           //串口模式 1，允许接收
TMOD|=0x20;          //初始化 T1 为定时功能,模式 2
PCON|=0x80;          //设置 SMOD=1
TL1=0xF4;            //波特率 4800bit/s，初值
TH1=0xF4;
IE|=0x90;            //中断
TR1=1;              //启动定时器

printf("_crol_(%bx,3)=%bx\n",a,_crol_(a,3)); //调用_crol_函数
printf("_lrol_(%d,3)=%d\n",b,_lrol_(b,3));   //调用_lrol_函数
printf("_lrol_(%ld,3)=%ld\n",c,_lrol_(c,3)); //调用_lrol_函数
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

_crol_(a5,3)=2d
_lrol_(13,3)=104
_lrol_(123456789,3)=987654312

```

在该程序中，首先声明并初始化变量，接着对串口进行初始化，然后分别调用_crol_函数、_lrol_函数和_lrol_函数来对 a、b 和 c 进行循环左移 3 位，结果将通过串口 0 输出。

12.6.2 循环右移函数

循环右移函数主要用于将数据按照二进制循环右移 n 位。按照操作数据类型的不同，其函数原型如下几种形式。

```

unsigned char _cror_(unsigned char val,unsigned char n);
unsigned int _iror_(unsigned int val,unsigned char n);
unsigned long _lror_(unsigned long val,unsigned char n);

```

其中，val 为待移位的变量，n 为循环移位的次数。函数_cror_、_iror_和_lror_分别用于字符型、整型和长整型变量的循环右移，其返回值分别为移位后的字符型、整型和长整型。该函数与 8051 单片机的 RRA 指令相关。程序示例如下：

```

#include <intrins.h>           //头文件
#include <stdio.h>
#include <reg51.h>

void main(void)              //主函数
{
    char a=0xA5;             //声明并初始化变量
    int b=13;
    long c=123456789;

    SCON=0x50;               //串口模式 1，允许接收
    TMOD|=0x20;              //初始化 T1 为定时功能,模式 2
    PCON|=0x80;              //设置 SMOD=1
    TL1=0xF4;                //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                //中断
    TR1=1;                   //启动定时器

    printf("_cror_(%bx,3)=%bx\n",a,_cror_(a,3)); //调用_crор_函数
}

```


51 单片机开发与应用技术详解

```
printf("_iror_(%d,3)=%d\n",b,_iror_(b,3));           //调用_iror_函数
printf("_lror_(%ld,3)=%ld\n",c,_lror_(c,3));         //调用_lror_函数
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
_cror_(a5,3)=b4
_iror_(13,3)=-24575
_lror_(123456789,3)=-1595180638
```

在该程序中，首先声明并初始化变量，接着对串口进行初始化，然后分别调用_cror_函数、_iror_函数和_lror_函数来对 a、b 和 c 进行循环右移 3 位，结果将通过串口 0 输出。

12.6.3 延时函数

延时函数用于使单片机程序产生延时，其函数原型如下：

```
void_nop_(void);
```

该函数类似于 8051 单片机的 NOP 指令。程序示例如下：

```
#include <intrins.h>           //头文件
#include <reg51.h>

void main(void)               //主函数
{
    P1=0xFF;                  //P1 端口输出 0xFF
    _nop_();                  //延时
    _nop_();
    P1=0x00;                  //P1 端口输出 0x00
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。在程序中，首先置 P1 端口为 0xFF，接着调用_nop_函数延时，然后置 P1 端口为 0x00。

12.6.4 位测试函数

位测试函数用于对字节中的一位进行测试，其函数原型如下：

```
void_testbit_(bit x);
```

其中，x 为待测位变量。如果该位变量置位则函数返回 1，同时将该位复位为 0，否则返回 0。该函数产生一个 8051 单片机的 JBC 指令，其只能用于可直接寻址的位，不允许在表达式中使用。程序示例如下：

```
#include <intrins.h>           //头文件
#include <stdio.h>
#include <reg51.h>

void main(void)               //主函数
{
    bit flag;                 //声明位变量

    SCON=0x50;                //串口模式 1，允许接收
    TMOD|=0x20;                //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                //设置 SMOD=1
    TL1=0xF4;                  //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                  //中断
    TR1=1;                     //启动定时器
```

```

flag=1;                                //置位变量
if(_testbit_(flag))                    //调用_testbit_函数
    printf("Bit was set\n");
else
    printf("Bit was clear\n");
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
Bit was set
```

在该程序中，首先声明并初始化位变量，接着对串口进行初始化，然后调用 `_testbit_` 函数测试该位变量是否置位，结果将通过串口 0 输出。

12.7 绝对地址访问函数

绝对地址访问的函数包含在头文件 `ABSACC.H` 中，是一些宏定义的函数。下面分别介绍常用的一些绝对地址访问函数。

12.7.1 BYTE 型存储空间访问函数

BYTE 型存储空间访问函数用于对 8051 单片机的存储空间进行绝对地址访问，可以做字节寻址。其宏定义原型如下：

```

#define CBYTE((unsigned char volatile code*)0)
#define DBYTE((unsigned char volatile idata*)0)
#define PBYTE((unsigned char volatile pdata*)0)
#define XBYTE((unsigned char volatile xdata*)0)

```

其中，`CBYTE` 用来寻址 CODE 区，`DBYTE` 用来寻址 IDATA 区，`PBYTE` 用来寻址 PDATA 区（可采用 `MOVX@R0` 指令），`XBYTE` 用来寻址 XDATA 区（可采用 `MOVX@DPTR` 指令）。程序示例如下：

```

#include <ABSACC.H>                    //头文件
#include <stdio.h>
#include <reg51.h>

void main(void)                        //主函数
{
    char xval;                          //声明变量

    DBYTE[0x0040]=0x45;                 //对 IDATA 区赋值
    PBYTE[0x0002]=0x45;                 //对 PDATA 区赋值
    XBYTE[0x0002]=0x45;                 //对 XDATA 区赋值

    xval=CBYTE[0x0002];                 //读取 CODE 区
    printf("xval=%bx\n",xval);
    xval=DBYTE[0x0040];                 //读取 IDATA 区
    printf("xval=%bx\n",xval);
    xval=PBYTE[0x0002];                 //读取 PDATA 区
    printf("xval=%bx\n",xval);
    xval=XBYTE[0x0002];                 //读取 XDATA 区
    printf("xval=%bx\n",xval);
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```

xval=5c
xval=45

```

```
xval=45
xval=45
```

在该程序中，首先声明并初始化位变量，接着对 IDATA、PDATA 和 XDATA 区赋值，最后分别读取 CODE、IDATA、PDATA 和 XDATA 区，结果将通过串口 0 输出。另外，通过使用 #define 预处理命令，可采用其他符号定义绝对地址，示例如下：

```
#define ABC XBYTE[0x1100]
```

该宏定义将符号 ABC 定义为外部数据存储器地址 0x1100。

12.7.2 WORD 型存储空间访问函数

WORD 型存储空间访问函数可以访问 8051 的所有存储器空间。其宏定义原型如下：

```
#define CWORD((unsigned int volatile code*)0)
#define DWORD ((unsigned int volatile idata*)0)
#define PWORD ((unsigned int volatile pdata*)0)
#define XWORD ((unsigned int volatile xdata*)0)
```

其功能与上一节的宏相似，只是其定义的数据类型为 unsigned int。程序示例如下：

```
#include <ABSACC.H> //头文件
#include <stdio.h>
#include <reg51.h>

void main(void) //主函数
{
    int xval; //声明变量

    DWORD[0x0040]=0x45; //对 IDATA 区赋值
    PWORD[0x0002]=0x45; //对 PDATA 区赋值
    XWORD[0x0002]=0x45; //对 XDATA 区赋值

    xval=CWORD[0x0002]; //读取 CODE 区
    printf("xval=%d\n",xval);
    xval=DWORD[0x0040]; //读取 IDATA 区
    printf("xval=%d\n",xval);
    xval=PWORD[0x0002]; //读取 PDATA 区
    printf("xval=%d\n",xval);
    xval=XWORD[0x0002]; //读取 XDATA 区
    printf("xval=%d\n",xval);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
xval=5924
xval=69
xval=69
xval=69
```

在该程序中，首先声明并初始化位变量，接着对 IDATA、PDATA 和 XDATA 区赋值，最后分别读取 CODE、IDATA、PDATA 和 XDATA 区，结果将通过串口 0 输出。

注意：使用该宏时，实际存储器地址应为 2*sizeof(unsigned int)。

12.7.3 far 存储区访问函数

far 存储区访问函数用于访问 far 存储器区域。其宏定义原型如下：

```
#define FVAR(object,addr)((object volatile far*)((addr)+0x10000L))
#define FCVAR(object,addr)((object const far*)((addr)+0x810000L))
```

其中，FVAR 用于访问 far 空间（存储类为 HDATA），FCVAR 用于访问 const far 空间（存

储类为 HCONST)。程序示例如下：

```
#include <ABSACC.H> //头文件

#define IOVAL FVAR(long,0x14FFE) //访问 HDATA 存储空间的地址 0x14FFE

void main(void) //主函数
{
    int var;
    var=IOVAL; //读
    IOVAL=0x10; //写
    var=FCVAR(int,0x24002); //从 HCONST 存储空间读地址 0x24002
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。程序中使用 FVAR 宏定义了 IOVAL，接着在主函数中使用 IOVAL 来进行访问 HDATA 存储空间的地址 0x14FFE。最后，通过 FCVAR 从 HCONST 存储空间读地址 0x24002。

注意 绝对地址目标不能超过 64KB 的段边界，例如不能在地址 0xFFEE 访问一个 long 变量。

12.7.4 far 存储区数组访问函数

far 存储区数组访问函数用于访问 far 存储器区域的数组类型目标。其宏定义原型如下：

```
#define FARRAY(object,base) *((object volatile far*) ((base)+0x10000L))
#define FCARRAY(object,base) *((object const far*) ((base)+0x810000L))
```

其中，FARRAY 用于访问 far 空间（存储类为 HDATA），FCARRAY 用于访问 const far 空间（存储类为 HCONST）。程序示例如下：

```
#include <ABSACC.H> //头文件
#define DPortRam FARRAY(int,0x24000)
void main(void) //主函数
{
    int i; //声明变量
    long l;
    l=FARRAY(long,0x8000)[i]; //使用 FARRAY 访问
    FARRAY(long,0x8000)[10]=0x12345678;
    DPortRam[i]= 0x1234;
    l= FCARRAY(long,0x81000)[5]; //使用 FCARRAY 访问
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。程序中分别演示了使用 FARRAY 和 FCRRAY 进行存储区读写的操作。

注意：绝对地址目标不能超过 64KB 的段边界，例如不能在起始地址为 0xFFFF9 访问一个有 20 个元素的 long 数组。

12.8 变量参数表函数

变量参数表函数用于函数参数的个数和类型可变的场合。C51 语言编译器允许函数的参数个数和类型是可变的，可使用简略形式（记号为“...”），这时参数表的长度和参数的数据类型在定义时是未知的。C51 的变量参数表函数包含在头文件 STDARG.H 中，主要包括如下几个。

➤ 函数原型 typedef char*va_list: 其功能为将 va_list 定义成指向参数表的指针。

- 函数原型 `void va_start (ap,v)`: 其功能为初始化 `ap` 参数, 其一般在一个可变长度参数表的函数中使用。在使用宏 `va_arg` 进行存取前, 必须调用函数来初始化可变参数表。
- 函数原型 `typedef va_arg (ap,type)`: 其功能为从 `ap` 指向的可变长度参数表中检索 `type` 类型的值。对 `va_arg` 的第一次调用将返回在 `va_start` 宏中指定的 `v` 参数后的第一个参数。继续调用 `va_arg` 将返回剩下的后续参数。对于每一个参数可以只调用一次, 而且必须按照参数表中的参数的次序进行。
- 函数原型 `void va_end (ap)`: 其功能为终止在 `va_start` 宏中已被初始化的可变长度参数表的指针 `ap`, 并且关闭参数表, 结束对可变参数表的访问。

在 C51 语言中, 定义具有可变参数的函数, 必须首先声明一个 `va_list` 型指针, 用 `va_start` 将指针初始化为指向给参数表, 用 `va_arg` 访问表中不同类型的参数, 对参数的访问结束后, 用 `va_end` 关闭参数表。下面举例讲解这些函数在具体程序设计中的应用, 程序示例如下:

```
#include <stdarg.h> //头文件
#include <stdio.h>
int varfunc (char *buf, int id, ...) //自定义函数
{
    va_list tag; //定义 tag 为参数表指针
    va_start (tag, id); //调用 va_start 函数
    if (id == 0)
    {
        int arg1;
        char *arg2;
        long arg3;

        arg1 = va_arg (tag, int); //调用 va_arg 函数返回参数
        arg2 = va_arg (tag, char *);
        arg3 = va_arg (tag, long);
    }
    else
    {
        char *arg1;
        char *arg2;
        long arg3;

        arg1 = va_arg (tag, char *); //调用 va_arg 函数返回参数
        arg2 = va_arg (tag, char *);
        arg3 = va_arg (tag, long);
    }
    va_end (tag); //关闭参数表
}

void main(void)
{
    char tmp_buffer [10];
    varfunc (tmp_buffer, 0, 27, "Test Code", 100L); //调用 varfauc 函数
    varfunc (tmp_buffer, 1, "Test", "Code", 348L);
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。程序中自定义了函数 `varfunc`, 其中分别调用变量参数表函数进行操作。在 `main` 主函数中, 则通过两次调用 `varfunc` 函数, 来实现不同参数的输入处理。

12.9 全程跳转函数

全程跳转函数用于正常系列函数的调用和函数结束, 还允许从深层函数调用中直接返回。其原型声明包含在头文件 `SETJMP.H` 中。下面介绍一些常用的全程跳转函数。

- 函数原型 `typedef char jmp_buf[jblen]`: 其功能为定义用于保存和恢复程序环境的缓冲

区，供 `setjmp` 和 `longjmp` 所使用，其必须定义为全局变量。

- 函数原型 `int setjmp (jmp_buf env)`：其功能为将程序执行的当前环境状态信息存入变量 `env` 中。当直接调用 `setjmp` 时，返回值为 0，当从 `longjmp` 调用时，返回值为非 0。
- 函数原型 `void longjmp (jmp_buf env, int retval)`：其功能为恢复先前调用 `setjmp` 时存在 `env` 中的环境状态信息，并从 `setjmp` 语句的下一条语句执行。参数 `val` 为调用 `setjmp` 的返回值。

在 C51 语言中，`longjmp` 和 `setjmp` 函数提供了一种执行非局部 `goto` 的方法。其一般用于执行控制传递给先前被调用的程序中的错误处理或恢复代码，一般只有用 `volatile` 属性声明的局部变量和函数参数才能被恢复。下面举例讲解这些函数在具体程序设计中的应用，程序示例如下：

```
#include <setjmp.h>                                //头文件
#include <stdio.h>

jmp_buf env;                                       //环境变量
bit error_flag;                                   //错误标志

void trig(void)
{
    if (error_flag != 0)
    {
        longjmp (env, 1);                          //向 setjmp 返回 1
    }
}

void recover (void)                                //恢复函数
{
}

void main (void)                                   //主函数
{
    if (setjmp (env) != 0)                          //首次调用 setjmp 将返回 0
    {
        printf ("LONGJMP called\n");
        recover ();                                  //调用恢复函数
    }
    else
    {
        printf ("SETJMP called\n");
        error_flag = 1;                             //错误标志
        trig ();                                     //调用 trig 函数
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
SETJMP called
LONGJMP called
```

程序中自定义了函数 `trig`。在 `main` 主函数中，首次调用 `setjmp` 将返回 0，从而执行 `trig` 函数。在 `trig` 函数中，通过 `longjmp` 向 `setjmp` 返回 1，则执行 `recover` 函数。

12.10 计算结构体成员的偏移量函数

计算结构体成员的偏移量函数包含在头文件 `STDDEF.H` 中。其函数声明如下：

```
int offsetof (structure, member);
```

51 单片机开发与应用技术详解

其中，`structure` 为结构体，`member` 为结构体成员。该函数计算 `member` 从开始位置的偏移量，并返回字节形式的偏移量值。程序示例如下：

```
#include<stddef.h>                                //头文件
#include <stdio.h>

struct Mystruct                                    //定义结构
{
    unsigned char type;
    unsigned long num;
    unsigned int len;
};

typedef struct Mystruct index;                    //声明 index

void main(void)                                    //主函数
{
    int x,y;
    x=offsetof(struct Mystruct,len);              //调用 offsetof，计算偏移量
    y =offsetof(index,num);                        //调用 offsetof，计算偏移量
    printf("x=%d,y=%d\n",x,y);                    //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，运行的结果如下：

```
x=5,y=1
```

程序中自定义了结构 `Mystruct`，在主函数中通过调用 `offsetof` 函数来计算偏移量，计算的结果通过串口输出。

12.11 小结

本章详细讲解了 C51 语言中常用的库函数，主要包括字符函数、字符串函数、I/O 函数、数学函数、标准函数、内部函数、绝对地址访问函数、变量参数表函数、全程跳转函数及偏移量函数。这些库函数涵盖了常见的字符、字符串、数学计算、I/O 控制等功能，读者在进行程序设计时可以直接调用使用。因此，熟练掌握和运用本章内容，可以大大减轻程序设计的负担，方便单片机的程序设计。

第 13 章 C51 语句和流程

C51 语句，即 C51 语言中的操作命令，用于使单片机完成特定的功能。C51 语言的源程序是由一系列的语句组成的，这些语句可以完成变量声明、赋值和控制输入输出等操作。一条完整的语句必须以“;”结束。由于单片机能识别的是机器指令，因此一般一条语句经过编译后生成若干条机器指令来执行。C51 语言中的语句包括说明语句、表达式语句、循环语句、条件语句、开关语句、复合语句、空语句和返回语句等，下面分别进行介绍。

13.1 说明语句

说明语句一般是用来定义声明变量，可以说明其类型和初始值。一般形式如下：

类型说明符 变量名 (=初始值);

其中，类型说明符指定变量的类型，变量名即变量的标示符，如果在声明变量的时候进行赋值，则需要使用“=”指定初始值。典型的说明语句示例如下，其中分别进行了变量声明及初始化赋值。

```
int a=1;           //声明并初始化整型变量
float c;           //声明浮点型变量
char p[6]="first"; //声明并初始化字符数组
sfr P1=0x80;       //声明并初始化寄存器
bit third;         //声明位变量
```

13.2 表达式语句

表达式语句是用来描述算术运算、逻辑运算或使单片机产生特定的操作。表达式语句是 C51 语言中最基本的一种语句。

示例如下：

```
b=b*20;
Count++;
X='A';Y='B';
P=(a+b)/a-1;
```

以上的都是合法的表达式语句。一般来说，任何表达式在末尾加上分号“;”，便可以构成语句。示例如下：

```
a=a+8           //赋值表达式
a=a+8;          //赋值语句
```

注意：这里需要强调表达式和语句的区别，一个程序的语句最后必须有分号。

表达式语句可以完成某种运算，但不进行赋值。示例如下：

a+b;

这也是一个语句，其使 a 和 b 完成相加运算，但不赋值给任何变量。虽然这个语句没有实际的任何意义，但它是一个合法的语句。

注意：不同的程序设计语言会有不一样的表达式语句，如 VB 中就是在表达式后面加入回车，就构成了 VB 的表达式语句，而在单片机的 C51 语言中则是加入分号“;”构成表达式语句。

13.3 复合语句

复合语句是用花括号“{}”将一组语句组合在一起而构成的语句。C51 语言中由单个表达式和末尾的分号构成的语句是简单语句。在 C51 语言中，复合语句是允许嵌套的，即就是在花括号“{}”中的“{}”也是复合语句。复合语句在程序运行时，“{}”中的各行单语句是依次顺序执行的。在 C51 语言中使用复合语句需要注意如下几点。

- C51 语言中，复合语句在语法上等同于一条单语句。
- 复合语句中不但可以由可执行语句组成，还可以用变量定义等语句组成。要注意的是在复合语句中所定义的变量，称为“局部变量”，所谓局部变量就是指它的有效范围只在复合语句中。
- 对于一个函数而言，函数体就是一个复合语句，函数内定义的变量有效范围只在函数内部。

复合语句在程序中的使用示例如下：

```
#include <stdio.h>                                     //头文件
void fun()                                              //函数定义
{
    int a=5,b=6;                                       //声明变量
    a=a-3;                                           //表达式语句
    b=b*2;
    printf("a=%d,b=%d\n",a,b);                       //输出结果
}

void main()                                           //主函数
{
    int a=21,b=22;                                    //声明变量
    printf("a=%d,b=%d\n",a,b);                       //输出结果
    {                                                //复合语句
        int a=3,b=4;                                  //声明变量
        a=a+1;                                         //表达式语句
        b=a+b;
        printf("a=%d,b=%d\n",a,b);                   //输出结果
    }
    printf("a=%d,b=%d\n",a,b);                       //输出结果
    fun();                                           //函数调用
    printf("a=%d,b=%d\n",a,b);                       //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

a=21,b=22

```
a=4,b=8
a=21,b=22
a=2,b=12
a=21,b=22
```

这段程序定义的变量 a 和 b 分别位于不同的复合语句中，从中可以看出其中变量之间的作用域关系。

典型的复合语句还有由 if、for 等构成的语句，这在后面分别介绍。

！注意：复合语句的构成符“{”和“}”之后均不能有分号。

13.4 循环语句

循环语句经常用于需要反复多次执行的操作。C51 语言中有三种基本的循环语句，即 while 语句、do-while 语句和 for 语句。这几个语句同样都是起到循环作用，但具体的作用和用法又不大一样。下面分别介绍。

13.4.1 while 语句

while 循环语句的一般使用形式如下：

```
while (表达式)
语句；
```

当其中的表达式值为真时，便执行语句，然后再次判断表达式的值，直到表达式的值为假时，才结束循环，并继续执行循环外的后续语句。

while 语句的特点是先判断条件，后执行语句。while 语句的循环过程如图 13-1 所示。

while 循环语句的示例程序如下：

```
#include <stdio.h>                                //头文件

void main()                                       //主函数
{
    int i=100,sum=0;                             //初始化
    while(i>0)                                   //while 循环
    {
        sum=sum+i;                               //累加
        i--;                                     //控制循环次数
    }
    printf("sum=%d\n",sum);                       //输出结果
}
```

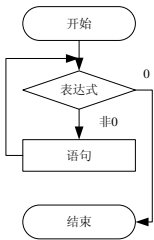


图 13-1 while 语句的循环过程

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
sum=5050
```

本例是求 1 加到 100 的总和，采用了 while 语句循环。当 i 的值等于 0 的时候，循环结束。使用 while 循环语句时，要注意以下几点。

- 如果语句是由多行构成，即是语句体，此时必须用“{}”括起来。即表示成复合语句的形式。
- while 语句循环体内允许空语句。

示例如下：

```
while((ch=getche())!='\X0D');
```

本例循环直到输入回车时，循环结束。

- while 语句循环允许多层循环嵌套。
- 需要在 while 循环的语句中或判断表达式中进行适当的条件修改，使其可以跳出循环，而不至于造成死循环。

13.4.2 do-while 语句

do-while 语句的一般形式如下：

```
do
语句;
while(表达式);
```

do-while 语句是先执行一次 do 后面的语句，然后再判断表达式是否为真，如果表达式为真，返回再次执行 do 后面的语句，直到表达式为假时，才结束循环，并继续执行循环外的后续语句。

do-while 语句的特点是先执行语句，后判断条件。因此，do-while 语句至少执行一次 do 后面的语句。同样，由多个语句构成语句体时，必须用“{}”括起来，表示成复合语句的形式。

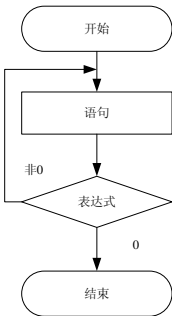


图 13-2 do-while 语句的循环过程

do-while 语句的循环过程如图 13-2 所示。程序示例如下：

```
#include <stdio.h> //头文件

void main() //主函数
{
    int i=100,sum=0; //初始化
    do //do-while 循环
    {
        sum=sum+i; //表达式语句
        i--; //表达式
    }while(i>0); //表达式
    printf("sum=%d\n",sum); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
sum=5050
```

本例是求 1 加到 100 的总和，采用了 do-while 语句循环。当 i 的值等于 0 的时候，循环结束。

注意：while (表达式) 后的“;”不能遗漏。另外使用本语句时不要构成死循环。

13.4.3 for 语句

for 语句的一般形式如下：

```
for(表达式 1;表达式 2;表达式 3)
语句;
```

其中，表达式 1 为赋值语句，给循环变量进行初始化赋值；表达式 2 是一个关系逻辑表达式，作为判断循环条件的真假；表达式 3 定义循环变量每次循环后按什么方式变化。当由表达式 1 初始化循环变量后，则由表达式 2 和表达式 3 可以确定循环次数。

求解完表达式 1 后，判断循环条件，即表达式 2 的真假，若条件为真，则执行下面的循环语句和表达式 3，直到循环条

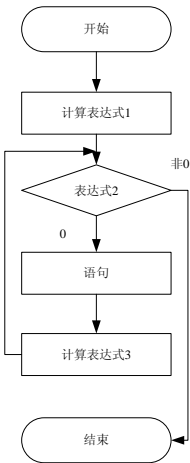


图 13-3 for 语句的循环过程

件为假时，才结束循环，然后继续执行循环外的后续语句。

for 语句的循环过程如图 13-3 所示。程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int i, sum=0;                                  //初始化
    for(i=0; i<=100; i++)                          //for 循环
    {
        sum=sum+i;                                //表达式语句
    }
    printf("sum=%d\n", sum);                        //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
sum=5050
```

本例是求 1 加到 100 的总和，采用了 for 循环语句。当 i 的值等于 0 的时候，循环结束。使用 for 循环语句时，需要注意以下几点。

- for 语句中的三个表达式都是可选择项，可以默认，但“;”不能省。例如，for(;;)表示一个无限循环。这里一定要注意不要构成一个死循环。
- 同 while 循环一样，for 语句循环允许多层循环嵌套。
- 语句可以是多个语句构成的语句体，但必须用“{ }”括起来。

13.5 条件语句

条件语句常用于需要根据某些条件来决定执行流向的程序中，由关键字 if 构成，即 if 条件语句。条件语句又被称为“分支语句”。C51 语言提供了三种形式的条件语句，下面分别进行介绍。

13.5.1 单分支结构

单分支结构的条件语句只有一个语句分支或语句块分支，其一般形式如下：

```
if (表达式) 语句;
```

其中，当 if 条件语句表达式的结果为真时，就执行分支语句，执行完后，继续执行后续程序；当表达式为假时，就跳过分支语句，执行后续程序。

使用 if 语句的单分支结构程序，示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    int a, b;                                       //变量声明
    a=1;                                           //初始化
    b=1;                                           //初始化
    if (a==b) a++;                                //if 语句的单分支结构
    printf("a=%d\n", a);                          //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

a=2

该程序中，首先声明变量 **a** 和 **b**，并进行初始化赋值，接着执行 **if** 语句的单分支结构。当 **a** 等于 **b** 时，就执行分支语句 **a++**。

13.5.2 双分支结构

双分支结构的条件语句包含两个语句分支，由关键字 **if** 和 **else** 构成，其一般形式如下：

```
if (表达式)
语句 1;
else
语句 2;
```

当表达式为真时，就执行语句 1，执行完后，继续执行 **if** 语句后面的语句；当表达式为假时，就执行语句 2，执行完后，继续执行 **if** 语句后面的语句。

使用 **if** 语句的双分支结构程序，示例如下：

```
#include <stdio.h>                                //头文件

void main()                                       //主函数
{
    int a,b;                                     //变量声明
    a=1;                                         //初始化
    b=2;                                         //初始化
    if (a==b)                                    //if 语句的双分支结构
        a++;
    else
        a--;
    printf("a=%d\n",a);                          //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

a=0

该程序中，首先声明变量 **a** 和 **b**，并进行初始化赋值，接着执行 **if** 语句的双分支结构。当 **a** 等于 **b** 时，就执行分支语句 **a** 加 1，否则执行 **a--**。

13.5.3 阶梯式 if-else-if 结构

阶梯式 **if-else-if** 结构是一种多分支结构，其可以包含多个分支语句，其一般形式如下：

```
if (表达式 1) 语句 1;
else if (表达式 2) 语句 2;
else if (表达式 3) 语句 3;
else if (表达式 n) 语句 n;
else 语句 n+1;
```

这是由 **if-else** 语句组成的嵌套，可以实现多方向条件分支。该语句从上到下逐个对条件进行判断，一旦条件为真，就执行与其相关的分支语句，并跳过剩余的阶梯；如果没有一个条件为真，则执行最后一个 **else** 分支语句 **n+1**。

在 C51 语言程序设计中采用阶梯式 **if-else-if** 结构时，需要注意以下几点。

- 在整个 **if** 语句中，只执行其中一条。
- **if** 和 **else** 是配对使用的，如果少了一个就会语法出错，**else** 总是与最临近的 **if** 相配对。

if-else-if 结构的执行流程如图 13-4 所示。这里举例说明这种结构在程序中的运用，程序示例如下：

```
#include <stdio.h>                                     //头文件

void main()                                             //主函数
{
    int score;
    char grade;
    score=87;                                           //初始化
    if(score>=95)                                       //进入多分支判断语句
        grade='A';
    else if(score>=80)
        grade='B';
    else if(score>=70)
        grade='C';
    else if(score>=60)
        grade='D';
    else
        grade='E';
    printf("score=%d,grade=%c\n",score,grade);        //输出结果
}
```

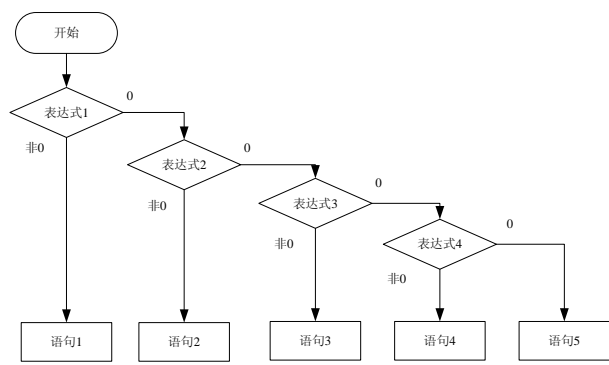


图 13-4 if-else-if 结构过程

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
score=87,grade=B
```

这段程序中，首先为 score 赋值，然后根据变量 score 的值来判断 grade 的等级，共有 5 个分支语句。

以上三种条件语句在 C51 语言程序设计中经常用到。使用这种 if 条件语句时，要注意以下几点。

- 分支语句可以是多个语句组成的语句体，此时必须用“{}”括起来。
- 条件语句循环允许多层条件进行嵌套。嵌套的一般形式如下：

```
if()
{
    if() 语句 1;
    else 语句 2;
else
{
    if() 语句 3;
    else 语句 4;
}
```

条件语句在进行嵌套时很容易出错，一般来说 else 语句与其上面最近的一个 if 语句相匹配。为了便于程序的理解，常用“{}”构成语句体来进行标记。推荐的书写形式如下：

```
if()
{
    if() 语句 1;
    else 语句 2;
}
else
{
    if() 语句 3;
    else 语句 4;
}
```

下面举一个条件语句嵌套的例子，程序示例如下：

```
#include <stdio.h>                                     //头文件

void main()                                             //主函数
{
    char ch;
    if(ch>='A' && ch<='z')
    {
        if(ch>='A' && ch<='Z')
            printf("A~Z");
        else
            printf("a~z");
    }
    else
        printf("No char");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行。读者可以从中领会如何使用条件语句嵌套，并养成良好的编程习惯。

13.6 开关语句

开关语句主要用于在程序中实现多个语句分支处理。在 C51 语言程序中，开关语句以关键字 switch 和 case 来标识。开关语句的一般形式如下：

```
switch(表达式)
{
    case 常量表达式 1:
        语句 1; break;
    case 常量表达式 2:
        语句 2; break;
    case 常量表达式 3:
        语句 3; break;
    case 常量表达式 n:
        语句 n; break;
    default:
        语句 n+1;
}
```

其中，关键字 switch 后面的表达式的值作为判断条件，与 case 后面的各个分支常量表达式的值相比较，如果相等时则执行对应的分支语句，再执行 break（间断语句）语句，跳出 switch 语句。如果 case 没有和条件相等的值时就执行关键字 default 后的语句。在程序中使用 switch 开关语句的应用示例如下：

```

#include <stdio.h>                                //头文件

void main()                                       //主程序
{
    char ch;
    ch='a';                                       //赋值
    switch(ch)                                    //开关语句
    {
        case 'a':                                //如果为'a',则输出'A'
            printf("ch 的大写字符为 A");
            break;
        case 'b':                                //如果为'b',则输出'B'
            printf("ch 的大写字符为 B");
            break;
        default:                                  //否则输出这里
            printf("Not a and b");
            break;
    }
}

```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

ch 的大写字符为 A

在程序中，首先声明字符型变量 **ch**，然后赋值字符“a”。此时，在使用 **switch** 开关语句进行判断时，执行第一个 **case** 分支语句，输出结果并执行 **break** 退出。

这里需要指出的是，使用多个 **if** 条件语句也可以实现多方向条件分支，可以起到 **switch** 开关语句同样的作用。但是使用过多的条件语句实现多方向分支，会使条件语句嵌套过多，程序冗长。这时使用开关语句既可以达到处理多分支选择的目的，又可以使程序结构清晰。

使用 **switch** 开关语句时，需要注意以下几点。

- **switch** 中的变量可以是数值，也可以是字符。
- 当要求没有符合条件的条件时不做任何处理，则可以不写 **default** 语句。
- 每个 **case** 和 **default** 后的分支语句可以是语句体，此时不需要使用“{}”括起来。

13.7 跳转语句

跳转语句主要用于程序执行顺序的跳转和转移。在 C51 语言中，跳转语句主要有三种，即 **goto** 语句、**break** 语句和 **continue** 语句。下面分别进行介绍。

13.7.1 goto 语句

goto 语句是一个无条件的转向语句，在 C51 语言程序执行到这个语句时，程序指针就会无条件地跳转到 **goto** 后的标号所在的程序段。**goto** 语句在很多高级语言中都有，其一般形式如下：

goto 语句标号；

其中的语句标号为一个带冒号的标识符。使用 **goto** 语句的程序示例如下：

```

#include <stdio.h>                                //头文件

void main()                                       //主函数
{
    int i=0,total=0;
    loop:                                       //语句标号

```


51 单片机开发与应用技术详解

```
total = total + i;           //执行运算
    i++;
    if(i<=100)               //如果满足条件则转向 loop 处
        goto loop;
printf("1+2+...+100=%d\n", total); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
1+2+...+100=5050
```

本程序用于求解 1 加到 100 的总和。程序定义了一个 loop 语句标号。当变量 i 的值小于等于 100 的时候，程序执行 goto 语句无条件跳转到 loop 处继续，实现求和运算。

在 C51 语言程序中使用 goto 语句进行程序设计时，需要注意如下几点。

- goto 语句后面标识符的定义应遵循前面章节所讲的 C51 标识符定义原则，不能用 C51 语言的关键字，也不能和其他变量和函数名相同，不然在程序编译时会出错。
- goto 语句不但可以无条件地转向，而且可以和 if 语句构成一个循环结构，上面的例子程序便是采用的这种方法。
- goto 语句可以用来跳出多重循环，不过只可以从内层循环跳到外层循环，而不能从外层循环跳到内层循环中。
- 在结构化程序设计时，对 goto 语句应当慎用，过多的使用会程序结构不清晰。因为 goto 语句破坏了程序模块化的结构，会使程序可读性变差、难于维护。

13.7.2 break 语句

break 语句通常用在循环语句和开关语句中，用来跳出循环程序块。其使用的一般形式如下：

```
break;
```

在 C51 语言程序设计中，break 语句主要用于如下两种情况。

- 当 break 用于开关语句 switch 中时，可使程序跳出 switch，而执行 switch 以后的语句。如果没有 break 语句，则 switch 语句将成为一个死循环而无法退出。
- 在 do-while、for、while 循环语句中时，break 语句和 if 语句联在一起使用，可以实现满足条件时便跳出循环的操作。

下面采用 while 语句来讲解 break 语句的使用，程序示例如下：

```
#include <stdio.h>           //头文件

void main()                  //主程序
{
    char ch[]={'a','b','c','d','e'}; //初始化字符数组
    int i=0;
    while(1)                  //循环按顺序查找字符
    {
        if(ch[i]=='c')         //如果找到则跳出 while
            break;
        i++;
    }
    printf("ch[%d]=%c\n",i,ch[i]); //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
ch[2]=c
```

这段程序中定义并初始化了字符数组 ch，然后通过 while 循环语句来逐个查找字符 c，当

查找到该字符时，便通过 **break** 语句跳出 **while** 循环并输出结果。

在 C51 语言程序设计中使用 **break** 语句时，要注意以下几点。

- **break** 语句只适用于单分支的条件语句，对 **if-else** 的条件语句不起作用。
- 如果遇到多层循环的情况，一个 **break** 语句只能向外跳出一层循环。

13.7.3 continue 语句

continue 语句的是用来跳过循环体中剩余的语句而强行执行下一次循环。其使用的一般形式如下：

```
continue;
```

在 C51 语言中，**continue** 语句只用在 **for**、**while**、**do-while** 等循环体中，常与 **if** 条件语句一起使用，可以提前结束本次循环。使用 **continue** 语句的程序示例如下：

```
#include <stdio.h>                                //头文件

void main()                                        //主函数
{
    char ch[]={'s','S','r','R','t'};              //初始化字符数组
    int i=-1;
    while(i<4)                                    //进入循环
    {
        i++;
        if(ch[i]>='A' && ch[i]<='Z')                //如果是大写字符则退出本次循环，进入下次循环
            continue;
        printf("ch[%d]=%c\n",i,ch[i]);            //输出小写字母
    }
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
ch[0]=s
ch[2]=r
ch[4]=t
```

在这段程序中，首先声明并初始化了一个字符数组，然后在 **while** 语句中判断每个字符是否是大写，如果是大写字符则使用 **continue** 语句退出本次循环，继续执行下次循环，否则将打印输出所有的小写字符。

13.8 函数调用语句

函数调用语句用于调用系统函数或用户自定义函数。在 C51 语言中，函数调用语句比较简单，在函数名后面加上分号便可构成函数调用语句。这里需要注意的是函数调用格式的问题，这将在后面的章节详细介绍。下面仅举一个例子加以说明。

```
#include <stdio.h>                                //头文件

void myprint()                                    //定义函数
{
    printf("hello world.\n");                    //输出字符串
}

int Add(int a)                                    //定义函数
{
    return a+1;                                  //返回值
}
```

```
void main()                                //主函数
{
    int i=2,j;                              //初始化
    myprint ();                             //调用函数语句
    j=Add(i);                               //调用带有返回值的函数语句
    printf("%d+1=%d\n",i,j);               //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
hello world.
2+1=3
```

从这段程序中可以看出，对于无返回值和参数传递的函数调用，直接在函数名后面跟上一组小括号和分号即可。如果需要参数传递，则应将参数传递写完整。

13.9 空语句

空语句是 C51 语言中又一个特殊的表达式语句，其仅由一个分号“;”组成。在实际程序设计时，有时为了语法的正确，要求有一个语句，但这个语句又没有实际的运行效果，那么这时就要有一个空语句。最典型的使用空语句的例子便是程序延时。

在 C51 程序中，while、for 构成的循环语句后面加一个分号，可以形成一个不执行其他操作的空循环体。常用来编写等待事件发生及延时的程序。示例如下：

```
#include <stdio.h>                          //头文件

void main()                                //主函数
{
    printf("First output");                 //输出字符串
    for (;a<50000;a++);                    //输出字符串
    printf("Delay some times and output");
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
First output
Delay some times and output
```

其中对于 for 语句来说，第一个分号是空语句，它会使 a 赋值为 0，但要注意的是如程序前有 a 值，则 a 的初值为 a 的当前值。第二个分号构成表达式语句。最后一个分号则使整个语句行成一个空循环。这样程序执行没有任何实际意义，唯一可以实现的是消耗 CPU 时间，常用来程序延时。

这里需要注意的是，在 C51 语言的头文件 intrins.h 中包含了一个空函数 nop()。该函数同样不执行任何有意义的操作，而是单纯地消耗 CPU 时间，使用 nop()函数便也可以完成空语句的功能。

13.10 返回语句

返回语句用于终止当前函数的执行，并强制返回到程序调用该函数的位置。在 C51 语言中，返回语句主要有以下两种形式。

```
return 表达式;

或者如下：
```

```
return;
```

其中，对于带有返回值的函数，则使用第一种返回语句，表达式的值便是函数的返回值。如果函数没有返回值，则可以缺省表达式，而采用第二种返回语句。返回语句在程序设计中的应用，示例如下：

```
#include <stdio.h>                                //头文件
int Add(int a,int b)                               //定义函数
{
    int c;
    c=a+b;                                         //加法运算
    return c;                                     //返回值
}

void main()                                       //主函数
{
    int i=2,j=3,k;                                //初始化
    k=Add(i,j);                                   //调用带有返回值的函数语句
    printf("k=%d\n",k);                           //输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

```
k=5
```

在程序中，首先定义了一个 Add 函数，用于计算两个整型数据的加法。在主函数中，调用该函数进行计算，通过 Add 函数的返回值得到加法的结果。

在 C51 语言程序设计中，使用返回语句需要注意如下几点。

- 函数内部可以有多个返回语句，但是程序仅执行其中的一个返回语句而退出函数。
- 一个函数的内部也可以没有返回语句，此时，程序执行到最后一个界定符“}”时自动返回。

13.11 C51 语言的流程控制结构

在 C51 语言的程序设计中，为了控制好模块间的顺序关系，一般采用的是模块化程序结构，这时需要采用一定的流程控制结构。C51 语言程序支持多种流程控制结构，比较常见的是顺序结构、分支结构和循环结构三种。

- 顺序结构的程序按代码的顺序自上而下执行，没有代码的跳跃。这种结构比较简单，常用于实现不是很复杂的任务。
- 选择结构的程序通过判断表达式的值来决定执行哪一段程序，一般采用条件语句 if、开关语句 switch 等来构成。这种结构常用于判断、决策等代码中。
- 循环结构的程序循环重复执行同一段代码，一般由 while、do-while、for 及 goto 等构成。这种结构常用于需要多次执行某项任务处理的场合，可以简化代码。

以上这三种程序结构，本质上是前面介绍的各个 C51 语句。在实际的单片机构程设计中，需要根据设计任务的特点，有选择地使用这些程序结构，来合理地简化并解决问题。对于比较复杂的程序设计，还可以多种流程控制结构嵌套使用，以达到最优的处理效果。

下面举一个比较复杂的例子，其中用到了多种语句来实现流程控制。这里要计算如下的数学表达式。

$$y = \begin{cases} x!(x > 0) \\ x^2 + x + 1 \quad (x \leq 0) \end{cases}$$

51 单片机开发与应用技术详解

其中 x 为整型数据，当 $x > 0$ 时， y 为 x 的阶乘；当 $x \leq 0$ 时， y 为 $x^2 + x + 1$ 的值。完整的程序代码示例如下：

```
#include <stdio.h>                                //头文件

int Fun(int x)
{
    int i,y;
    if(x>0)                                        //判断 x 的值
    {
        y=1;
        for(i=1;i<=x;i++)                        //阶乘运算
        {
            y=y*i;
        }
    }
    else
    {
        y=x*x+x+1;                               //计算数学表达式
    }
    return y;                                     //返回语句
}

void main()                                       //主函数
{
    int x=5,y;
    y=Fun(x);                                    //函数调用
    printf("y=%d\n",y);                          //打印输出结果
}
```

该程序可以在 Keil μ Vision3 编译环境中执行，其运行的结果如下：

y=120

这里首先定义了一个函数 **Fun** 用于处理上面的分段数学表达式。函数内部采用 **if** 条件语句实现分段处理，对于阶乘运算则采用了 **for** 循环语句，最后通过 **return** 返回语句返回运算的结果。主函数中通过函数调用语句执行运算并将结果输出。

13.12 小结

本章详细讲述了单片机 C51 语言中的各种语句结构，包括说明语句、表达式语句、复合语句、循环语句、条件语句、开关语句、跳转语句、函数调用语句、空语句和返回语句。这些语句是 C51 语言程序的重要组成部分。灵活利用这些语句，可以实现不同的程序流程控制结构，如顺序结构、选择结构和循环结构等，以达到简化程序的目的。熟练掌握本章内容是进行 C51 语言程序设计的基础。

第 14 章 预处理及用户配置文件

预处理命令通常在程序编译时进行一些符号处理，其并不执行具体的硬件操作。C51 语言中的预处理命令主要有宏定义指令、文件包含指令和条件编译指令，还有其他一些调试时使用的指令。本章将详细介绍各种预处理命令及 C51 语言的用户配置文件，并结合一定的程序实例以加深读者的理解。

14.1 预处理命令概述

C51 语言中提供了各种预处理命令，类似于汇编程序中的伪指令。一般来说，在对源程序进行编译前，C51 语言编译器需要先对程序中的预处理命令进行处理，然后将预处理的结果与源程序一并进行编译，最后产生目标代码。通过这些预处理命令，在很大程度上为 C51 语言提供功能和符号等方面的扩展，使用预处理命令也可以提高程序的可读性。

为了与源程序中的语句相区别，预处理命令前要加一个“#”。C51 语言程序中的预处理命令包括以下几个。

#define	//用于宏定义
#error	//用于程序调试
#include	//用于文件包含
#if	//用于条件编译
#else	//用于条件编译
#elif	//用于多种条件编译选择
#endif	//用于条件编译
#ifdef	//用于条件编译
#ifndef	//用于条件编译
#undef	//用于宏定义
#line	//用于更改行号
#pragma	//用于传送控制指令

按照使用的范围，这些预处理命令可以分为如下几大类，在下面几节将分别对其进行介绍。

- 文件包含指令：#include；
- 宏定义指令：#define、#undef；
- 条件编译指令：#if、#else、#ifdef、#ifndef、#endif；
- 其他编译指令：#line、#error、#pragma。

14.2 宏定义指令

宏定义指令是用一些标识符作为宏名来代替一些符号或常量的命令。宏定义指令可以带参

数，也可以不带参数。下面分别介绍用于宏定义的一些预处理指令。

14.2.1 #define 命令

#define 命令用于定义一个“宏名”。其中“宏名”是一个标识符，在源程序中遇到该标识符时，均以定义的串的内容替代该标识符。ANSI 标准将标识符定义为“宏名”，这个替换过程称为“宏替换”。#define 命令用于定义宏名时，可以带参数，也可以不带参数，下面分别介绍这两种情况。

1. 不带参数的宏定义

不带参数的宏定义，其一般形式如下：

```
#define 标识符 字符串
```

其中，#define 是宏定义指令，标识符即宏名，字符串是被替换的对象。典型的宏定义指令示例如下：

```
#define TURE      1
#define FALSE     0
#define PI        3.1415926
```

执行这些宏定义指令后，在源程序编译过程中，如果遇到 TURE 就用 1 代替，FALSE 就用 0 代替，PI 就用 3.1415926 代替。

为了便于程序的阅读和修改，在使用宏定义指令时，应注意以下几点。

- 宏定义指令应该放到文件的开始处，而不是将其分散到整个程序中。
- 如果宏定义指令较多，可将其放到独立的文件中，并用#include 指令来访问。
- 为了与变量名相区别，一般使用大写字母定义标识符，这样便于发现宏替换的位置。
- 在指令的标识符和字符串之间可以有任意个空格或制表符。
- 如果所要替换的字符串长于一行，为了便于程序的阅读，可以在该行末尾用一反斜杠“\”续行。示例如下：

```
#define LONG_STR  "this is a very long string that is used as an \
example and do not make any mistakes"
```

- 宏定义不是语句，所以不要在后面加分号，如果加了分号，程序编译时，会将分号作为字符串的一部分，从而一并进行替换。
- 宏定义的宏名可以嵌套使用。当宏名定义后，即可成为其他宏名定义中的一部分。示例如下：

```
#define ONE      1
#define TWO      ONE + ONE
#define THREE    ONE + TWO
```

这里首先宏定义了 ONE 来替代 1，接着在宏定义 TWO 时，使用两个 ONE 的和来替代 1+1=2。宏定义 THREE 时，采用相同的方法。

- 在源程序中，用引号括起来的字符串不发生宏替换。示例如下：

```
#include <stdio.h>                                //头文件
#define HB  "Happy Birthday!\n"                   //宏定义

void main()                                         //主函数
{
    printf(HB);                                    //输出宏定义字符串
    printf("HB");                                  //输出字符串
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

```
Happy Birthday!
HB
```

➤ 宏定义可以用来表示数组的大小，这样便于程序的修改。示例如下：

```
#define SIZE 10
int ch[SIZE];
```

这里首先宏定义 **SIZE** 为 10，然后声明了一个整型数组 **ch**，其大小由 **SIZE** 来指定。这样，如果修改宏定义 **SIZE** 的大小，则数组的大小也就改变了。

2. 带参数的宏定义

带参数的宏定义指令，其一般形式如下：

```
#define 宏名(参数表) 字符串
```

其中，**#define** 是宏定义指令，标识符即宏名，参数表是形参列表，字符串是被替换的对象。在程序每次遇到宏名时，与之相连的形参均由程序中的实参代替。示例如下：

```
#define FUN(A,B) 2*A*B
y=FUN(3,5);
```

这里首先宏定义了 **FUN**，在程序中使用 **FUN** 时，将括号中的实参 3 和 5 分别替换形参 **A** 和 **B**，即该表达式等同于下面的表达式

```
y=2*3*5;
```

带参数的宏定义和函数的作用很相似。用宏替换代替函数可以增加代码的执行速度，因为不存在函数的调用过程，但是由于重复编码会使最终程序代码的长度增加。下面举例说明带参数宏定义的使用，程序示例如下：

#include <stdio.h>	//头文件
#define PI 3.14	//宏定义 PI=3.14
#define L(R) 2*PI*R	//带参数的宏定义，用于计算周长
#define S(R) PI*R*R	//带参数的宏定义，用于计算面积
 void main()	 //主函数
{	
printf("L=%f\n",L(10));	//输出周长
printf("S=%f\n",S(10));	//输出面积
}	

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

```
L=62.800000
S=314.000000
```

在本例中，首先宏定义 **PI** 替代浮点型数据 3.14，接着采用带参数的宏定义来定义周长和面积的计算。在主程序中，分别调用这两个宏定义来求解半径为 10 的圆的周长和面积。

14.2.2 #undef 命令

#undef 命令用于取消前面已定义过的宏名。一般形式如下：

```
#undef 宏名
```

其中，宏名为前面用 **#define** 定义过的标识符。使用 **#undef** 的目的是将宏名局限在仅需要的代码段中。示例如下：

```
#include <stdio.h>                                     //头文件
```



```
#define COUNT 25                                //宏定义

void main()                                     //主函数
{
    printf("COUNT =%d\n", COUNT);              //输出 COUNT =10
    #undef COUNT                                //撤销宏定义
    //printf("COUNT =%d\n", COUNT);            //此时再引用是错误的
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

```
COUNT =25
```

在该程序中，`#undef COUNT` 之前，`COUNT` 是有定义的，其值为 25。`#undef COUNT` 之后，`COUNT` 便没有定义，因此不能引用，该输出语句是错误的。

14.3 文件包含指令

文件包含指令`#include` 通常在 C51 语言程序的开头，将另外一文件的内容引入当前文件。其中被包含的文件通常是头文件、宏定义等，利用文件包含指令可以有助于更好地调试文件。其一般形式如下：

```
#include "头文件.h"
#include <头文件.h>
#include 宏定义标识符
```

其中`#include` 为文件包含指令，双引号或尖括号括起来的是读入的源文件。典型的文件包含指令示例如下：

```
#include "myfile.h"
#include <math.h>
define MATH_FILE "C\keil\inc\math1.h"
#include MATH_FILE
```

下面举例讲解文件包含指令在程序设计中的应用。程序示例如下：

```
#include <stdio.h>                                //头文件
#include "myfile.h"                                //包含文件

void main()                                       //主函数
{
    int i,sum=0;
    for(i=0;i<=10;i++)                            //主循环
    {
        sum=sum+fun(i);
    }
    printf("sum=%d\n",sum);                        //打印输出结果
}
```

其中 `myfile.h` 文件的内容如下：

```
int fun(int i)                                    //自定义函数
{
    int mul;
    mul=i*I;                                       //乘积运算
    return mul;                                   //返回结果
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

```
sum=385
```

本程序中，在文件 `myfile.h` 中自定义了函数 `fun` 用于计算乘积。在主函数文件中，通过文件包含指令将该文件读入。此时便可以调用 `fun` 函数来进行计算。

使用文件包含指令 `#include` 有利于程序的模块化设计。将常用的函数做成文件，在编写其他程序时，如果要用到这些函数，只需将其包含进来即可。

在使用 `#include` 命令时，要注意以下几点。

- 在 C51 语言程序中，一个 `#include` 命令只能包含一个文件。
- `#include` 命令出现在哪，被包含的文件就在哪里引入。一般来说，被包含的文件要放在包含文件的头部。
- 被读入的源文件用双引号括起来时，表示先从源文件所在目录搜索，如果未发现文件，再到系统指定的目录搜索。
- 被读入的源文件用尖引号括起来时，表示直接到系统指定的目录搜索，这种方式一般用于包含库文件的头函数。例如 `#include<math.h>`。
- 当文件为自定义文件时，编译器通常先搜索当前目录，其次按 `#include` 指定的目录去检索。
- 如果文件标识符中包含文件路径，则仅在指定路径中搜索被嵌入文件。

14.4 条件编译指令

条件编译指令用于对程序源代码的各部分有选择地进行编译。采用条件编译，可以提高程序的适用性，缩小目标代码的大小。

在默认情况下，源程序中的所有行都要进行编译。但是有时需要某些语句行在条件满足的情况下，才进行编译，此时便用到条件编译指令。目前商业软件公司广泛应用条件编译来制作某个程序的许多不同用户版本。

14.4.1 `#if`、`#else`、`#endif` 命令

`#if`、`#else`、`#endif` 指令用于条件编译的一般形式如下：

```
#if 常数表达式
    语句段;
#else
    语句段;
#endif
```

其中，`#if`、`#else`、`#endif` 为条件编译指令，常数表达式为判断的条件，语句段为条件编译部分。执行过程为，如果常量表达式为真，则编译其后面的语句段；如果常量表达式为假，则编译 `#else` 后面的语句段；`#endif` 命令是一个条件编译的结束。下面举例说明条件编译语句在程序设计中的应用，示例如下：

```
#include <stdio.h>                                //头文件
#define SCORE 85                                    //宏定义

void main()                                         //主函数
{
    #if SCORE>=60                                    //开始条件汇编
        printf("You have passed the exam!\n");
    #else
        printf("You lost the exam!\n");
```

```
#endif                                     //条件编译结束
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：
You have passed the exam!

- 在使用 #if、#else、#endif 条件编译指令时，需要注意以下几点。
- 跟在 #if 后面的表达式必须仅含常量及已定义过的标识符，不可使用变量。
 - 条件编译的结束指令 #endif 后面没有分号 “;”。
 - 常量表达式不许含有操作符 sizeof。

14.4.2 #elif 命令

#elif 命令用于进行多种编译选择。其意义与 “else if” 相同，形成一个 if-else-if 阶梯状语句。#elif 命令条件编译的一般形式如下：

```
#if 表达式 0
    语句段;
#elif 表达式 1
    语句段;
#elif 表达式 2
    语句段;
#elif 表达式 3
    语句段;
...
#elif 表达式 n
    语句段;
#endif
```

其中，#elif 后跟一个常量表达式，如果表达式的值为真，则编译其后的代码块，不对其他 #elif 表达式进行测试。否则，顺序测试下一块。使用 #elif 命令用于进行多种编译选择的程序示例如下：

```
#include <stdio.h>                               //头文件
#define SCORE 85                                   //宏定义

void main()                                       //主函数
{
    #if SCORE>=90                                //开始条件编译
        printf("Very good!\n");
    #elif SCORE>=80
        printf("Good!\n");
    #elif SCORE>=60
        printf("You only passed the exam!\n");
    #else
        printf("You lost the exam!\n");
    #endif                                       //条件编译结束
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：
Good!

本例中，首先宏定义分数 SCORE 为 85，然后在主程序中通过 #elif 命令进行多种编译选择，并输出该分数的等级分类。

在 C51 语言中，`#if` 与 `#elif` 命令可以一直嵌套到实现规定的权限，其中 `#endif` 或 `#else` 与最近的 `#if` 或 `#elif` 相关联。

14.4.3 `#ifdef`、`#ifndef` 命令

`#ifdef` 与 `#ifndef` 命令用于判断宏名是否被定义，并根据判断的情况进行条件编译。`#ifdef` 命令的一般形式如下：

```
#ifdef 宏名
    语句段;
#endif
```

其执行过程是，如果宏名在前面 `#define` 语句中已定义过，则后面的语句段将被编译。`#ifndef` 的一般形式如下：

```
#ifndef 宏名
    语句段;
#endif
```

其执行过程是，如果宏名在 `#define` 语句中无定义，则后面的语句段将被编译。下面举例讲解 `#ifdef` 和 `#ifndef` 命令在程序中的应用，示例如下：

```
#include <stdio.h>                                //头文件
#define SCORE 85                                   //宏定义

void main()                                        //主函数
{
#ifdef SCORE                                       //条件编译
    printf("SCORE existed!\n");                 //如果宏 SCORE 存在，则执行该语句
#else
    printf("No SCORE!\n");                       //如果宏 SCORE 不存在，则执行该语句
#endif

#ifndef GREED
    printf("GREED is not defined!\n");           //如果宏 GREED 存在，则执行该语句
#else
    printf("GREED is defined!\n");               //如果宏 GREED 不存在，则执行该语句
#endif
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

```
SCORE existed!
GREED is not defined!
```

在本例中，首先定义了宏名 `SCORE`，接着在主程序中通过 `#ifdef` 和 `#ifndef` 命令来判断宏名 `SCORE` 和 `GREED` 是否被定义过，并根据情况打印输出结果。

在 C51 语言中，使用 `#ifdef` 和 `#ifndef` 命令需要注意如下几点。

- 可以像嵌套 `#if` 那样，`#ifdef` 与 `#ifndef` 也支持嵌套。
- `#ifdef` 与 `#ifndef` 可以与 `#else` 语句联合使用，但不能和 `#elif` 联合使用。

14.5 其他编译指令

除了以上几种预处理指令外，在 C51 语言中还定义其他几种编译预处理指令，主要用于编译和调试程序等。下面分别进行讲解。

14.5.1 #line 命令

#line 命令用于改变_LINE_与_FILE_的内容。其中_LINE_和_FILE_是在编译程序中预先定义的标识符，分别表示行号和源文件。#line 命令使用的一般形式如下：

```
#line 数字["文件名"]
```

其中的数字为任意正整数，可选的文件名为任意有效文件标识符。行号为源程序中当前行号，文件名为源文件的名字。命令#line 主要用于调试及其他特殊应用。下面举例讲解#line 命令在程序中的使用，示例如下：

```
#include <stdio.h>                                //头文件
#line 200                                           //初始化行计数器
void main()                                       //行号 200
{                                                 //行号 201
printf("行号=%d\n", __LINE__);                  //行号 202
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

```
行号=202
```

本例中，首先使用#line 命令初始化行计数器，然后在主程序中输出当前语句的行号。printf 语句显示数 202，因为它是语句#line100 后的第 3 行。

14.5.2 #error

#error 命令用于强迫编译程序停止编译，主要用于程序调试。其使用的一般形式如下：

```
#error "message"
```

其中，message 为错误消息。下面举例讲解#error 命令在程序设计中的应用，示例如下：

```
#include <stdio.h>                                //头文件
#define SCORE 85                                  //宏定义

void main()                                       //主函数
{
#ifdef SCORE                                     //条件编译
printf("SCORE existed!\n");                     //如果宏 SCORE 存在，则执行该语句
#else
#error "No SCORE!\n"                            //如果宏 SCORE 不存在，则执行该处
#endif

#ifdef GREED
#error "GREED is not defined!\n"                //如果宏 GREED 存在，则执行该处
#else
printf("GREED is defined!\n");                  //如果宏 GREED 不存在，则执行该语句
#endif
}
```

本例中，分别对宏名 SCORE 和 GREED 是否定义过进行判断，如果 SCORE 没有定义过则停止编译，并输出错误信息 “No SCORE!”，如果如果 GREED 没有定义过则停止编译，并输出错误信息 “GREED is not defined!”。这里程序在编译时的输出信息，如图 14-1 所示。

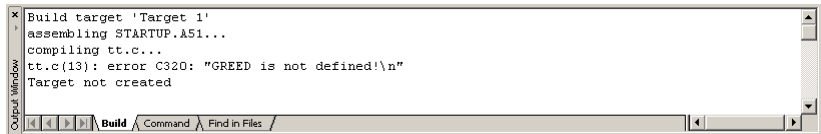


图 14-1 编译输出错误信息

14.5.3 #pragma

#pragma 命令用于向编译程序传送各种 C51 语言编译器的控制指令。根据#pragma 指令后面的字符串，编译系统将按照特定的方式来翻译 C51 语言的字符串和函数。

#pragma 指令后面的字符串，可以大写，也可以小写。示例如下：

```
#pragma sfr                                //在 C51 语言中使用 SFR
#pragma access                             //使用绝对地址
#pragma asm                                //在 C51 语言中插入汇编语句
```

下面举例说明#pragma 命令在程序设计中的应用，示例如下：

```
#include <REG51.h>                        //头文件

void main()                               //主函数
{
    while(1)
    {
        P1=0x00;                          //P1 端口输出
        #pragma asm                       //嵌入汇编语句
            NOP
            SETB P1.7                      //置 P1.7 为高电平
            NOP
        #pragma endasm                   //结束汇编语句嵌入
    }
}
```

该程序可以在 Keil μ Vision3 集成开发环境中运行。在本程序中，首先在 P1 端口输出 0，然后通过#pragma 命令嵌入汇编语句，用于置 P1.7 为高电平。

在 C51 语言中使用#pragma 命令，需要注意如下几点。

- C51 语言编译器的控制指令只能在源文件的开始，使用#pragma 命令设定。
- 同一指令在一个源文件中不能使用两次以上。
- 如果编译系统不支持#pragma 指令，那么将忽略该指令而继续向下编译。

14.6 C51 语言编译器的控制指令

C51 语言编译器的控制指令分为源文件控制类、目标文件控制类及列表控制类三类。这些指令可能需要具体的编译器支持，因此这里仅进行简单的说明，用户可以参考具体的编译器。对于 Keil μ Vision3 集成开发环境，可以在项目选项中进行设置。下面分别介绍各类中常用的控制指令。

14.6.1 源文件控制类

比较常用的有如下两个，具体介绍可以参加相应的编译器。

- NOEXTEND：C51 源文件不允许使用 ANSI C 扩展功能；
- DEFINE (DF)：定义预处理（在 C51 命令行）。

14.6.2 目标文件（Object）控制类

比较常用的有如下几个，详细内容需要参加所使用的编译器。

- COMPACT LARGE SMALL: 选编译模式;
- DEBUG (DB): 包含调试信息, 以供仿真器或 dScope51 使用;
- NOAMAKE (NOAM): 禁止 AutoMake 信息记录;
- NOREGPARMS: 禁止用寄存器传递参数;
- OBJECTTEXTEND(OE) Object: 文件包含附加变量类型信息;
- OPTIMIZE (OT): 指定优化级别;
- REGFILE (RF): 指定一个寄存器使用的文件以供整体优化用;
- REGISTERBANK (RB): 指定一个供绝对寄存器访问的寄存器区名;
- SRC: 不生成目标文件只生成汇编源文件。

14.6.3 列表文件（Listing）控制类

比较常用的有如下几个，详细内容需要参加所使用的编译器。

- CODE (CD): 向列表文件加入汇编列表;
- LISTINCLUDE (LC): 显示 include 文件;
- SYMBOLS (SB): 列表文件包括模块内所有符号的列表;
- WARNINGLEVEL (WL): 选择“警告”级别。

14.7 C51 语言的用户配置文件

C51 语言的用户配置文件包括启动代码文件、变量初始化文件、基本 I/O 函数文件、分组配置文件, 这些文件都是以源程序的形式存放在 Keil\C51\LIB 文件夹中。用户根据需要适当修改配置文件可以满足不同的硬件环境需要。

C51 编译器在对用户创建的项目进行编译连接时, 会自动将上述代码添加到用户程序中去。需要修改配置文件时, 可以通过 Keil μ Vision3 的项目窗口, 将需要修改的配置文件添加到自己的文件组中, 并在编辑窗口进行修改, 然后进行总体编译连接, 这样就可以将修改后的配置文件代码连接到自己的程序代码中。

14.7.1 C51 语言启动代码文件

C51 语言启动代码文件用于在主程序进入 main 函数之前, 完成对 8051 单片机片内外 RAM 清零、开设常规堆栈及再入函数堆栈、设置堆栈指针等任务。Keil μ Vision3 编译器针对不同类型的 8051 单片机提供了多种启动代码配置文件, 其中最常用的启动代码文件是 STARTUP.A51, 而其他的启动代码都和其基本类似。

启动代码文件 STARTUP.A51 的主要功能包括如下几个方面。

- 定义内部 RAM 大小、外部 RAM 大小、可重入堆栈位置。
- 清除内部、外部或以此页为单元的外部存储器。
- 按存储模式初使化重入堆栈及堆栈指针。
- 初始化 8051 硬件堆栈指针。
- 向 main 函数交权。

在 C51 语言项目建立的时候, 由连接定位器 BL51 自动将启动代码文件加入到 C51 语言源程序代码前面。启动文件 STARTUP.A51 的目标代码已经驻留在 C51 语言编译器的运行库中, 只要单片机执行复位操作, 则该文件将立即被执行。

在启动文件 `STARTUP.A51` 中包含一些 EQU 语句，如表 14-1 所示。可以根据需要，通过修改这些语句的值，从而对系统进行初始化设置。

表 14-1 STARTUP.A51 中的数据含义

常 数 名	意 义
IDATALEN	待清内部 RAM 长度
XDATASTART	指定待清外部 RAM 起始地址
XDATALEN	待清外部 RAM 长度
IBPSTACK	是否小模式重入堆栈指针需初始化标志，1 为需要，默认为 0
IBPSTACKTOP	指定小模式重入堆栈顶部地址
XBPSTACK	是否大模式重入堆栈指针需初始化标志，默认为 0
XBPSTACKTOP	指定大模式重入堆栈顶部地址
PBPSTACK	是否 Compact 重入堆栈指针，需初始化标志，默认为 0
PBPSTACKTOP	指定 Compact 模式重入堆栈顶部地址
PPAGEENABLE	P2 初始化允许开关
PPAGE	指定 P2 值
PDATASTART	待清外部 RAM 页首址
PDATALEN	待清外部 RAM 页长度

由于启动代码的重要性，下面给出 Keil C51 的 `STARTUP.A51` 文件的源代码，如下所示。可以看到文件是用汇编语言来编写的，用户可以根据需要进行改写。原文是英文注释的，为了阅读和理解方便，下面对其中的重要部分进行了中文注释。

```
$NOMOD51
;-----
;  STARTUP.A51: 用户上电进行初始化的程序.
;
;   A51  STARTUP.A51
;-----
;
;定义单片机上电初始化的内存空间
;
;使用 EQU 命令定义
;
IDATALEN      EQU      80H      ;初始化 IDATA 存储器的字节数
;
XDATASTART    EQU      0H      ;初始化 XDATA 存储器的绝对起始地址
XDATALEN      EQU      0H      ;初始化 XDATA 存储器的字节数
;
PDATASTART    EQU      0H      ;初始化 PDATA 存储器的绝对起始地址
PDATALEN      EQU      0H      ;初始化 PDATA 存储器的字节数
;
; 注意: 在物理硬件上, IDATA 存储器空间包括了 8051 系列单片机中的 DATA 存储器空间和 BIT 存储器空间
;-----
;
; 模拟初始化再入堆栈
;
```


51 单片机开发与应用技术详解

```
; 用 EQU 指令初始化再入函数堆栈指针
;
;单片机存储器模式选用 SMALL 模式时，再入函数的堆栈空间
IBPSTACK      EQU      0                ;使用 SMALL 存储器模式的再入函数时，需要将其设置为 1
IBPSTACKTOP    EQU      0FFH+1          ;将堆栈顶设置为最高地址+1
;
;单片机存储器模式选用 LARGE 模式时，再入函数的堆栈空间
XBPSTACK      EQU      0                ;使用 LARGE 存储器模式的再入函数时，需要将其设置为 1
XBPSTACKTOP    EQU      0FFFFH+1        ;将堆栈顶设置为最高地址+1
;
;单片机存储器模式选用 COMPACT 模式时，再入函数的堆栈空间
PBPSTACK      EQU      0                ;使用 COMPACT 存储器模式再入函数时，需要将其设置为 1
PBPSTACKTOP    EQU      0FFFFH+1        ;将堆栈顶设置为最高地址+1
;
;-----
;
; 单片机使用 COMPACT 存储器模式时，64KB XDATA 存储器空间的分页定义
;
;用 EQU 指令定义 PDATA 类型变量在 XDATA 存储器空间中的页地址
;使用 EQU 指令定义 PPAGE 时必须与 L51 连接定位器 PDATA 指令的控制参数一致
;
PPAGEENABLE    EQU      0                ;使用 PDATA 类型变量时，需要将其设置为 1
;
PPAGE          EQU      0                ;定义页号
;
PPAGE_SFR      DATA    0A0H            ;为 SFR 提供最高地址比特
; (8051 变量使用 P2 作为 i 最高地址)
;
;-----

;标准 SFR 符号
ACC      DATA    0E0H
B        DATA    0F0H
SP       DATA    81H
DPL      DATA    82H
DPH      DATA    83H

NAME      ?C_STARTUP      ;模块名为?C_STARTUP

?C_C51STARTUP  SEGMENT  CODE      ;指名代码段
?STACK        SEGMENT  IDATA      ;指名堆栈段

RSEG      ?STACK      ;指名堆栈
DS        1

EXTRN CODE (?C_START) ;指名单片机主程序的起始地址
PUBLIC ?C_STARTUP

CSEG      AT          0          ;定义用户程序的起始地址，一般在 MON51 仿真器中使用
?C_STARTUP:  LJMP      STARTUP1

RSEG      ?C_C51STARTUP

STARTUP1:
```

;单片机在上电时清零 IDATA 内存。如果不需要上电清零 IDATA，可以注销 IF 到 IFEND

;之间的语句，或者修改 IDATALEN 的长度。为了让 CPU 具有掉电保护功能，需要确定 IDATALEN 的长度

```
;
IF IDATALEN <> 0
    MOV     R0,#IDATALEN - 1
    CLR     A
IDATALOOP:    MOV     @R0,A
    DJNZ    R0,IDATALOOP
ENDIF
```

;单片机在上电时 XDATA 内存清零。如果不需要上电清零 XDATA，可以修改 XDATALEN 的长度

;或者注销 IF 到 IFEND 之间的语句

```
IF XDATALEN <> 0
    MOV     DPTR,#XDATASTART
    MOV     R7,#LOW (XDATALEN)
    IF (LOW (XDATALEN)) <> 0
        MOV     R6,#(HIGH (XDATALEN)) +1
    ELSE
        MOV     R6,#HIGH (XDATALEN)
    ENDIF
    CLR     A
XDATALOOP:    MOVX    @DPTR,A
    INC     DPTR
    DJNZ    R7,XDATALOOP
    DJNZ    R6,XDATALOOP
ENDIF
```

;送 PDATA 存储器页面高位地址

```
IF PPAGEENABLE <> 0
    MOV     PPAGE_SFR,#PPAGE
ENDIF
```

;单片机在上电时 PDATA 内存清零。如果不需要上电清零 PDATA，可以修改 PDATALEN 的长度

;或者注销 IF 到 IFEND 之间的语句

```
IF PDATALEN <> 0
    MOV     R0,#LOW (PDASTART)
    MOV     R7,#LOW (PDATALEN)
    CLR     A
PDATALOOP:    MOVX    @R0,A
    INC     R0
    DJNZ    R7,PDATALOOP
ENDIF
```

;设置指名使用 SMALL 存储器模式时再入函数的堆栈空间

```
IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)

    MOV     ?C_IBP,#LOW IBPSTACKTOP
ENDIF
```

;设置指名使用 LARGE 存储器模式时再入函数的堆栈空间

```
IF XBPSTACK <> 0
EXTRN DATA (?C_XBP)

    MOV     ?C_XBP,#HIGH XBPSTACKTOP
    MOV     ?C_XBP+1,#LOW XBPSTACKTOP
ENDIF
```

;设置单片机使用 COMPACT 存储器模式时，再入函数的堆栈空间

```
IF PBPSTACK <> 0
EXTRN DATA (?C_PBP)
```

```
MOV      ?C_PBP,#LOW PBPSTACKTOP
ENDIF

;设置单片机堆栈的起始地址
MOV      SP,#?STACK-1
;如果程序超过 64kB，使用了程序分组技术，则启动下面的程序
; EXTRN CODE (?B_SWITCH0)
; CALL    ?B_SWITCH0           ;初始化代码分组 0
;程序从第一组开始执行
;使用 LJMP 指令跳转到用户的主程序 main 函数处
LJMP     ?C_START
END
```

14.7.2 C51 语言启动代码分析

前面介绍了 C51 语言的启动代码，下面详细介绍上面的启动代码文件中常用的一些 EQU 语句的功能。

1. IDATALEN

IDATALEN 用于声明 8051 单片机开始运行时，需要清零的内部 RAM 字节数。在启动代码文件中，IDATALEN 一般默认值为 80H。而对于 8052 单片机，由于其具有 256 字节内部 RAM，因此可设置为 100H。

一般来说，只有当程序需要在开始时清零内部 RAM，才有必要对 IDATALEN 进行赋值。如果希望单片机内部 RAM 具有掉电保护功能，则应将 IDATALEN 设置为 0。

2. XDATASTART、XDATALEN

XDATASTART 和 XDATALEN 用于声明需要清零的 XDATA 储存器的首地址和长度。其中 XDATASTART 指出对应的 XDATA 储存器的首地址，XDATALEN 指出需要进行清零 0 的字节数。

3. PDATASTART、PDATALEN

PDATASTART 和 PDATALEN 用于声明需要清零的 PDATA 储存器的首地址和长度。其中 PDATASTART 指出对应的 PDATA 储存器的首地址，PDATALEN 指出需要清零的字节数。

4. IBPSTACK、IBPSTACKTOP

IBPSTACK 和 IBPSTACKTOP 用于定义在 SMALL 编译模式下创建的再入函数的模拟栈区。其中，IBPSTACK=1 时表示创建模拟栈，并对栈指针(变量 C_IBP)进行初始化；IBPSTACK=0 (默认值)时表示不创建模拟栈。IBPSTACKTOP 用于指出 SMALL 编译模式下再入函数的模拟堆栈区栈顶首地址，默认值为 IDATA 区的 0XFF 地址。

Keil μ Vision3 编译器对于该栈不检查其是否能满足特定要求，用户需要自己进行判断。

5. XBPSTACK、XBPSTACKTOP

XBPSTACK 和 XBPSTACKTOP 用于定义在 LARGE 编译模式下创建的再入函数的模拟栈区。其中 XBPSTACK=1 时表示创建模拟栈，并对栈指针(变量 C_XBP)进行初始化；XBPSTACK=0 (默认值)时表示不创建模拟栈。XBPSTACKTOP 则指出 LARGE 编译模式下再入函数的模拟堆栈区栈顶首地址，默认值为 XDATA 区的 0XFFF 地址。

Keil μ Vision3 编译器对于该栈不检查其是否能满足特定要求，用户需要自己进行判断。

6. PBPSTACK、PBPSTACKTOP

PBPSTACK 和 PBPSTACKTOP 用于定义在 COMPACT 编译模式下创建的再入函数的模拟

栈区。其中 PBPSTACK=1 时表示创建模拟栈，并对栈指针（变量 C_PBP）进行初始化；XBPSTACKTOP 指出栈顶首地址，默认值为 PDATA 区的 0XFFFF 地址。

Keil μ Vision3 编译器对于该栈不检查其是否能满足特定要求，用户需要自己进行判断。

7. PPAGEENABLE、PPAGE

PPAGEENABLE 和 PPAGE 用于在 COMPACT 编译模式下,使用“页寻址”方式操作 PDATA 存储区域。对于 LARGE 编译模式，使用这些指令可以提高程序的运行速度或减少程序代码的长度。

PPAGEENABLE 允许对 8051 单片机 P2 进行初始化，以实现 对存储器空间的“页寻址”操作。例如，需要初始化 P2 作为 COMPACT 模式高端地址，则必须置 PPAGEENAGLE=1，PPAGE 为 P2 值，如果指定某页 1000H~10FFH，则 PPAGE=10H，而且连接时必须如下设置。

```
L51<input modules> PDATA(1080H) ; 其中 1080H 是 1000H~10FFH 中的任一个值。
```

PPAGEENABLE 和 PPAGE 必须与连接定位器 BL51 的控制指令“PDATA”一起使用。其中，PDATA 指令是用来指定 XDATA 存储器中 PDATA 区的首地址。

Keil μ Vision3 编译器和 BL51 连接定位器都不检查 PPAGE 和 PDATA 指令的正确与否，用户必须自行保证 PPAGE 和 PDATA 为合理的值。

除了 STARTUP.A51 启动代码文件外，还有一些用于特殊系列单片机的启动代码文件，如 START751.A51、STARTLPC.A51、START390.A51、START_AD.A51 等。它们的使用方法与启动代码文件 STARTUP.A51 类似，只是增加了一些特殊功能，如设置片内数据存储器 SRAM 的地址范围、扩展堆栈寻址方式、调整 CPU 时钟、端口复位状态、看门狗定时器、片内 EPROM 的保密状态、扩展堆栈模式及堆栈空间大小等。

14.7.3 变量初始化文件

变量初始化文件用于对源程序中声明的变量进行初始化赋值。在 Keil μ Vision3 编译环境中，主要的变量初始化文件为 INIT.A51，此外还提供了 INIT_TNY.A51 文件用于不使用外部 XDATA 存储器的实时系统。

当 C51 语言源程序中包含有初始值的外部变量和静态变量时,连接定位器 BL51 将会自动将该目标代码加入到 C51 语言源程序的前面，对已确定初始化的外部变量和静态变量进行赋值。这是因为变量初始化文件的目标代码已经驻留在 Keil μ Vision3 的编译器的运行库内。

如果需要对初始化文件进行修改，可以将其添加到用户自己的项目文件中，修改完毕后重新对项目进行整体编译即可。

INIT.A51 文件包含一个看门狗定时器的宏定义 WATCHDOG，如果用户源程序使用了看门狗定时器，并且对变量初始化处理所需的时间比看门狗定时器的刷新时间要长。此时宏定义 WATCHDOG 中必须包含看门狗刷新时间的代码。

例如，用户采用 AT89S52 单片机中的看门狗定时器，则可在 INIT.A51 文件开始处加入以下语句。

```
T3                EQU      0FFH                //看门狗定时器的地址
PCON              EQU      087H                //特殊功能寄存器 PCON 的地址
WATCH_TNTV       EQU      156                //监视间隔为 200ms
```

将 WATCHDOG 宏定义修改为如下：

```
WATCHDOG          MACRO
```

```
ORL      PC0, #10H           //置位 PCON.4
MOV      T3,      # WATCH_TNTV //写入时间间隔
ENDM
```

然后将修改后的文件添加到项目中进行编译即可。

除了 INIT.A51 文件和 INIT_TNY.A51 文件外,还有一些用于专门系列单片机的变量初始化文件,如 INIT751..A51、INIT_MX.A51 等,它们的使用方法与 INIT.A51 文件类似。

14.7.4 基本 I/O 函数文件

- 基本 I/O 函数文件有 PUTCHAR.C 和 GETKEY.C 两个。
- PUTCHAR.C 用于将字符串从串行口输出。其采用 XON/XOFF 协议进行控制,将换行字符“LF”(\\n)被转换为“CR, LF”(\\r\\n)。在 Keil μ Vision3 编译器中,是 printf, puts 等函数的字符输出核心函数。用户可以根据自己的需要来修改该文件,用以实现特定的输出效果,例如 LCD 或 LED 显示等。
 - GETKEY.C 用于字符的串口输入。其不进行数据转换。在 Keil μ Vision3 编译器中,是 C51 语言编译器运行库中的 getchar, scanf 等函数的字符输入核心函数。用户可以根据自己的需要来修改该文件,用以实现特定的输入效果,例如做矩阵键盘的输入等。

14.7.5 分组配置文件

分组配置文件用于对用户程序进行代码分组设计。在 Keil μ Vision3 编译器中提供了两个分组配置文件,下面分别介绍。

由于传统的 8051 单片机只能将 ROM 扩展到最大 4MB,而新一代扩展型单片机可将其存储器范围扩展到最大 16MB。为此提供了 L51_BANK.A51、XBANKING.A51 等分组配置文件。

1. L51_BANK.A51 文件

L51_BANK.A51 文件用于传统 8051 单片机的分组代码设计。其 ROM 最大只能扩展到 4MB,在对有 C51 语言编译器生成的浮动代码目标文件进行连接定位时,应采用 BL51 连接定位器。

2. XBANKING.A51 文件

XBANKING.A51 文件用于新一代 80C51 单片机的分组代码设计。其支持 far 和 far const 存储器类型变量,扩展连接定位器 Lx51 通过 far 和 far const 存储器类型来访问扩展 HDATA 和 HCONST 存储器地址空间。

Lx51 连接定位器允许在 16MB 的物理 code 空间和 16MB 的 xdata 空间中分别定位 HCONST 类和 HDATA 类。一般来说,用 far 存储器类型定义的变量存放在 HDATA 存储类中,而用 far const 存储器类型定义的变量存放在 HCONST 存储类中。C51 语言编译器通过三个字节的一般指针访问这类存储器区域。

在使用新型 8051 单片机进行大容量存储器扩展设计时,如果使用了 far 和 far const 类型的变量,则需要根据需要修改 XBANKING.A51 文件,然后再对整个项目进行编译连接,并最终生成目标代码。

14.8 小结

本章详细介绍了 C51 语言所支持的各种预处理命令,包括宏定义指令、文件包含指令、条

件编译指令和其他一些编译指令。然后又介绍了 C51 语言编译器的一些控制指令，这需要和具体的编译器相联系。最后还介绍了一下 C51 语言的用户配置文件。熟练掌握本章内容，对读者以后的单片机程序设计有很大帮助。

第 15 章 C51 语言的存储结构

C51 语言是面向 51 系列单片机的硬件控制系统开发语言，其与单片机的硬件资源有着密切的联系。C51 语言的程序经过编译后，是要在单片机上运行的。因此，程序代码及数据都是需要按照一定的存储类型保存在 8051 单片机的特定存储区中的。存储结构的组织形式是单片机 C51 语言很重要的组成部分。本章将介绍 C51 语言的存储结构、存储类型和动态内存分配函数等相关内容。

15.1 存储器结构

存储器结构是存储器的组织形式。51 系列单片机采用哈佛结构，将程序代码存储器（ROM）和数据存储器（RAM）分开，各自遵循各自的寻址机构和寻址方式。下面概括一下 51 系列单片机存储器的硬件构成。

15.1.1 51 系列单片机的存储区域

51 系列及其兼容单片机在物理上划分为如下 4 个存储区。

➤ 片内数据存储器区、片外数据存储器区、片内程序存储器区、片外程序存储器区

早期的单片机片内存储器比较小，最近新推出的 51 内核单片机在存储器方面有了很大的扩展。每个存储区的大小和用法，可以参阅不同型号单片机的说明。

15.1.2 片内数据存储器（RAM）的结构

C51 语言程序中的变量一般保存在片内数据存储器，这样取址速度快。当然变量也可以保存在片外数据存储器，这将在后面进行介绍。51 系列单片机内部数据存储器（RAM）可划分为如下两大区域。

➤ 00H~7FH：片内低 128 字节 RAM 区

➤ 80H~0FFH：特殊功能寄存器区（SFR）

对于地址为 00H~07FH 的低 128 字节片内 RAM 区，又可划分为如下三个区域。

1. 通用寄存器区

通用寄存器区的地址为 00H~1FH，共由 4 个寄存器组，分别为第 0 组（00H~07H）、第 1 组（08H~0FH）、第 2 组（10H~17H）、第 3 组（18H~1FH）。每个寄存器组含有 8 个通用寄存器，即 R0、R1、R2、R3、R4、R5、R6、R7。这样总共有 32 个通用寄存器，每个都可以用寄存器名寻址，也可以直接用字节地址寻址，地址为 00H~1FH。

2. 可位寻址区

8051 系列单片机片内 RAM 的可位寻址区地址为 20H~2FH，共 16 个字节单元，这些单元可按字节寻址，也可按位寻址，共 128 位。

3. 用户 RAM 区

8051 系列单片机片内 RAM 的用户 RAM 区地址为 30H~7FH。这个区域只能用字节地址

寻址，堆栈可以设置在该区。

15.2 存储类型

存储类型是指程序中变量和常量等在 8051 硬件系统中的存放方式。Keil μ Vision3 编译器完全支持 8051 系列单片机的硬件结构，可完全访问 8051 单片机硬件系统的所有部分，并将数据定位在不同的硬件存储区中。

单片机 C51 语言中支持的存储类型有 data、bdata、idata、pdata、xdata、code 几种。在 8051 系列单片机的硬件系统中，存储类型与存储区关系，如表 15-1 所示。

表 15-1 存储类型与存储区关系

存储类型	与存储区的关系
data	直接寻址片内数据存储区，访问速度快（128 字节）
bdata	可位寻址片内数据存储区，允许位与字节混合访问（16 字节）
idata	间接寻址片内数据存储区，可访问片内全部 RAM 地址空间（256 字节）
pdata	分页寻址外部数据存储区（256 字节），由 MOVX@R0 访问
xdata	可寻址片外数据存储区（64KB），由 MOVX@DPTR 访问
code	程序代码存储区（64KB），由 MOVC@DPTR 访问

下面分别讲解这几种存储类型。

15.2.1 data 存储类型

将变量设置为 data 存储类型的一般形式如下：

标识符 data 变量名

其中，data 为关键字，标识符为变量的类型。在 51 单片机系统中，对 data 区寻址是最快的。所以一般来说，应将频繁使用的变量设置为 data 型。示例如下：

```
int      data ch[5];
char     data string[6];
float    data flo;

MyType   data newtype;                // 自定义的数据类型
```

除了程序变量可以设置在 data 区外，堆栈和寄存器也可以声明在 data 区。由于 data 区只有 128 字节，因此要注意节约使用，而且要定义足够大的堆栈空间，防止堆栈溢出。

15.2.2 bdata 存储类型

将变量设置为 bdata 存储类型的一般形式如下：

标识符 bdata 变量名

其中，bdata 为关键字，标识符为变量的类型。bdata 区是可位寻址数据存储器，可以将要求可位寻址的数据定义为 bdata。示例如下：

```
unsigned char bdata ie;                // 在位寻址区定义字符变量 ie
int           bdata mn[10];           // 在位寻址区定义整型数组 mn[10]
sbit          ie5=ie^5;               // 用关键字 sbit 定义位变量来可寻址对象其中一位
sbit          mn8=mn[1]^8;
```

首先将 ie 和 mn[10] 设置为 bdata 存储类型，接着定义 sbit 类型的变量 ie5 和 mn8，其值由位寻址 ie 和 mn 来获得。

需要注意的是，在使用 bdata 定义变量时，不允许定义 float 和 double 型变量。若要对实型数寻址，可用包含 float 和 long 的联合来实现。示例如下：

```
typedef mytype                // 定义联合
```



```

{
unsigned long count;           //定义无符号长整型，32 位
float ff;                      //定义浮点型，32 位
}bitf;                          //联合名
bitf bdata usefloat;           //在 bdata 段声明联合
sbit flo=usefloat^31;          //定义位变量名

```

15.2.3 idata 存储类型

将变量设置为 idata 存储类型的一般形式如下：

标识符 idata 变量名

其中，idata 为关键字，标识符为变量的类型。idata 类型为间接寻址片内数据存储区，可访问片内全部的 256 字节 RAM 地址空间。示例如下：

```
extern float idata tt;          //在位寻址区定义浮点型变量 tt
```

注意：idata 区可以定义浮点型变量。

15.2.4 pdata 存储类型

将变量设置为 pdata 存储类型的一般形式如下：

标识符 pdata 变量名

其中，pdata 为关键字，标识符为变量的类型。pdata 类型存储在分页寻址外部数据存储区，示例如下：

```
unsigned charpdata pun;        //在位寻址区定义字符变量 pun
```

15.2.5 xdata 存储类型

将变量设置为 xdata 存储类型的一般形式如下：

标识符 xdata 变量名

其中，xdata 为关键字，标识符为变量的类型。对 xdata 区寻址，须装入 16 位地址，因此尽量将外部数据存放在 xdata 区。示例如下：

```

#include <stdio.h>              //头文件
#include <reg51.h>              //头文件
unsigned char pdata RX1;       //声明 xdata 变量
unsigned char pdata RX2;       //声明 xdata 变量

void main()                    //主函数
{
while(1)
{
    RX1=P1;                    //赋值
    RX2=P3;                    //赋值
}
}

```

15.2.6 code 存储类型

将变量设置为 code 存储类型的一般形式如下：

标识符 code 变量名

51 单片机开发与应用技术详解

其中，code 为关键字，标识符为变量的类型。示例如下：

```
char code text[]="Hello everyone!";           //在 code 区定义字符串数组
```

使用 code 存储类型定义数据时，C51 语言编译器会将其定义在程序代码存储器（ROM 或 EPPROM）。

在使用上面几种存储类型时，因为访问内部数据存储器比访问外部数据存储器快，所以应将频繁使用的变量放在内部 RAM 中，将很少使用的变量放在外部 RAM 中。

15.3 扩展数据类型

扩展数据类型是 8051 硬件和 C51 语言编译器所特有的，不属于 ANSI C 标准。这些类型的数据可以操作 8051 的特殊功能寄存器，而不能用指针对其进行存取。扩展数据类型可由以下几种关键字说明。

➤ sfr: 用于字节寻址，定义 8 位特殊功能寄存器。示例如下：

```
sfr P0=0x80;           //为 P0 口地址为 80H 后的一个字节
sfr P1=0x90;           //为 P1 口地址为 90H 后的一个字节
```

➤ sfr16: 用于字寻址，定义 16 位特殊功能寄存器，且该 16 位必须低位在低字节，高位在紧跟的高字节才行。示例如下：

```
sfr16 T2=0xCC;
该语句用于指定 52 系列单片机 Timer2 入口地址，其中 T2L=0xCC，紧跟着 T2H=0xCD。
```

➤ sbit: 用于位寻址，从位寻址字节中定义位变量。例如：

```
sbit EA=0xAF;
该语句用于指定第 0xAF 位为 EA，即中断允许。
```

➤ bit: 定义位变量，其值为 0 或 1。示例如下：

```
bit flag;
flag=1;
该语句定义了一个位变量 flag，接着为该变量赋值 1。
```

以上扩展类型的数据将使用 8051 的 128 个字节的 SFR 寻址区，该区域可位寻址、字节寻址或字寻址，用以控制定时器、计数器、串口、I/O 及其他一些硬件资源。在头文件 reg51.h 中，大量使用了这些扩展数据类型，用户可以查阅其中的描述。下面分别进行讲解。

15.3.1 sfr 和 sfr16

sfr 和 sfr16 可以用于定义 8051 的特殊功能寄存器，其一般形式如下：

```
sfr      特殊功能寄存器名=特殊功能寄存器地址常数
sfr16    特殊功能寄存器名=特殊功能寄存器地址常数
```

其中，sfr 和 sfr16 为关键字。特殊功能寄存器的定义示例如下：

```
sfr      P2=0xA0;           //定义 P2 的 I/O 端口，其地址为 A0H
sfr16    T2=0xCC;           //定义定时器 2，其地址 T2L=CCH，T2H=CDH
```

在 C51 语言中，使用 sfr 和 sfr16 定义变量要注意以下几点。

- 关键字后的变量名要符合标识符的命名规则，并且含义明确，这样可以提高程序的可读性。
- 用 sfr 定义 8 位特殊功能寄存器时，等号后面必须是常数，不允许有带运算符的表达式，且该常数必须位于特殊功能寄存器的地址范围之内（80H~FFH）。
- sfr 用来定义 8 位特殊功能寄存器，sfr16 用来定义 16 位特殊功能寄存器。
- sfr16 不能用于定时器 0 和 1 的定义。

- 用 sfr16 定义 16 位特殊功能的寄存器时，等号后面是其低地址，其相应的高地址一定要位于物理低位地址之上。

15.3.2 sbit

sbit 用于定义可位寻址对象，例如特殊功能寄存器某位。在 C51 语言中，有如下几种定义方法。

- sbit 位变量名=位地址。该语句用于将位地址赋值给位变量名，示例如下：

```
sbit P1_1=0x91; //将位的绝对地址赋给位变量
```

这里需要注意的是，sbit 的位地址必须位于 80H~FFH 之内。

- sbit 位变量名=特殊功能寄存器名^位位置。该语句使用符号“^”来获得位数据，并赋值给位变量。示例如下：

```
sfr P1=0x90; //定义一个特殊功能寄存器名
sbit P1_1=P1^1; //指定位变量名所在的位置
```

- sbit 位变量名=字节地址^位位置。该语句和第二种方法是一样的，只是将特殊功能寄存器名用位地址常数直接表示。示例如下：

```
sbit P1_1=0x90^1;
```

15.3.3 bit 型变量

bit 型变量可用于变量类型、函数声明、函数返回值等场合，其定义的一般形式如下：

```
bit 变量名
```

采用 bit 定义的位变量存放于内部 RAM（20H~2FH）。位变量在 C51 语言里是外部变量，但编译系统不对其初始化，必须在程序中初始化位变量。示例如下：

```
extern bit bch;
```

在 C51 语言中，使用 bit 定义变量，需要注意以下几点。

- 用 #pragma disable 说明的函数和用 unsigned 指定的函数，不能返回 bit 值。
- 位变量不能作为结构体和联合体的成员。
- 位变量不能作为函数的参数。
- 位变量不能作为自动变量和函数内的静态变量。
- 和位变量运算的操作数只能是 0 和 1。
- 运算符*、&、sizeof 不能用于位变量。
- 一个 bit 变量不能声明为指针，例如 bit *ptr 是错误的。
- 不能定义 bit 型数组，例如 bit arr[5]是错误的。

15.4 存储模式

存储模式是编译环境中指定的变量的存储区形式。一般来说，如果在定义变量时默认存储类型，编译系统将按照存储模式所规定的默认存储区来指定变量、函数参数等的存储区域。Keil μ Vision 编译系统支持的 8051 系列单片机存储模式共有如下三种。

15.4.1 Small 模式

Small 模式用于将所有未指明存储区的变量均装保存在内部 RAM，即采用 data 存储类型方式。采用 Small 模式的优点是访问速度快，缺点是空间有限，需要节约使用存储空间，只适用于规模较小的程序。

15.4.2 Compact 模式

Compact 模式用于将所有未指明存储区的变量均装保存在外部 RAM 区的一页（256B）内，即采用 pdata 存储类型方式。

采用 Compact 模式的优点是存储空间较 Small 模式宽裕，缺点是访问速度要慢些，但较下面介绍的 Large 模式要快，是一种中间状态。Compact 模式适用于变量不超过 256 字节，可以通过 P2 口指定地址的高字节。

15.4.3 Large 模式

Large 模式用于将所有未指明存储区的变量放在多达 64KB 的外部 RAM 区中，即采用 xdata 存储类型方式。采用 Large 模式的优点是空间大，可存变量多，缺点是速度较慢。Large 模式使用数据指针 DPTR 来对变量进行寻址。

15.4.4 存储模式的选择

存储模式一般是在 C51 编译器选项中进行选择，也可以在程序中指定。在程序中指定存储模式的形式如下：

```
void fun1(void) small{ }
```

该语句声明函数 fun1，用 small 说明函数内部变量全部保存在内部 RAM。对于一些经常使用的和特别耗时的地方可以这样声明，有利于提高运行速度。

15.5 C51 语言的存储器指针

在 C51 语言中支持一般指针和存储器指针。对变量进行声明时可以指定变量的存储类型。同样，在对于指针变量声明的时候，也可以指定存储类型。下面分别进行介绍。

15.5.1 一般指针

一般指针的声明在前面已经介绍过，不过同时还可以说明指针的存储类型。示例如下：

```
long * state; //state 为指向 long 型整数的指针，则依存储模式存放
char *xdata ptr; //ptr 为一个指向 char 数据的指针，放于外部 RAM 区
```

一般指针可存放于任何存储器中，其用三个字节存放，分别为存储器类型、高位偏移、低位偏移量。使用一般指针时，可以访问数据而不用考虑其在存储器的位置，因此十分方便。示例如下：

```
#include <stdio.h> //头文件
#include <reg51.h>

void main() //主函数
{
    char *cp; //定义一般指针
    int *ip;
    long *lp;

    char data dc='A'; //定义在 data 存储器中
    int data di=12;
    long data dl=1979;

    char xdata xc='B'; //定义在 xdata 存储器中
    int xdata xi=23;
    long xdata xl=1980;

    char code cc='C'; //定义在 code 存储器中
```

```

int code ci=34;
long code cl=1981;

cp=&dc; //一般指针可以访问 data 存储器
ip=&di;
lp=&dl;
printf("%c %d %ld\n",*cp,*ip,*lp); //输出结果

cp=&xc; //一般指针可以访问 xdata 存储器
ip=&xi;
lp=&xl;
printf("%c %d %ld\n",*cp,*ip,*lp); //输出结果

cp=&cc; //一般指针可以访问 code 存储器
ip=&ci;
lp=&cl;
printf("%c %d %ld\n",*cp,*ip,*lp); //输出结果
}

```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

```

A 12 1979
B 23 1980
C 34 1981

```

从该程序中，可以看出这种指针一般可以完全访问 data、xdata 和 code 存储器。

15.5.2 存储器指针

存储器指针在声明时指定了存储类型，它总是指向特定的存储区。示例如下：

```

char data *str; //str 指向 data 区中 char 型数据
int xdata *pow; //pow 指向外部 RAM 的 int 型整数

```

由于存储器的类型已经指定了，因此这种指针存放时，只需存放偏移量即可。对于 idata、data、bdata 和 pdata 存储类型只需一个字节，对于 code 和 xdata 存储类型需要两个字节就够了。下面举例讲解存储器指针在程序中的应用，示例如下：

```

#include <stdio.h> //头文件
#include <reg51.h>

void main() //主函数
{
    char data *cp; //指向 data 存储器中的 char 型变量的指针变量
    int xdata *ip; //指向 xdata 外部存储器中的 int 型变量的指针变量
    long code *lp; //指向 code 存储器中的 long 型变量的指针变量

    char data dc='A'; //初始化
    int xdata buf1[3]={'A',12,2008};
    long code lx=2008;

    cp=&dc; //赋值
    ip=&buf1[1];
    lp=&lx;

    printf("%c %d %ld\n",*cp,*ip,*lp); //输出结果
}

```

该程序可以在 Keil μ Vision3 集成开发环境中运行，其运行的结果如下：

A 12 2008

在程序中，首先定义存储器指针，接着进行初始化赋值，最后打印输出结果。

15.5.3 指针存储类型与指针所指向的数据的存储类型的关系

指针存储类型与指针所指向的数据的存储类型是不同的概念，在使用时一定要注意。示例如下：

```
uchar xdata tmp[10];           //在外 RAM 区占 10 个字节的内存空间 ( 0x0000 ~ 0x0009 )
uchar data * data pstr;
pstr=tmp;
```

其中 tmp 保存在外部 RAM，访问外 RAM 需要两个字节来寻址 64KB 空间，而 pstr 使用 data 关键字，所以编译器将其编译成指向内部 RAM 的指针变量了。最后的赋值语句便是错误的。上面的语句可以改为如下形式。

```
uchar xdata tmp[10];
uchar xdata * data pstr;
pstr = tmp;
```

这样是指在内部 RAM 分配一个指针变量 pstr，而且这个指针本身指向 xdata 区。此时便可以进行访问。这种情况效率最高的访问外 RAM 的方法。上面的语句还可以改为如下形式。

```
uchar xdata tmp[10];
uchar xdata * xdata pstr;
pstr=tmp;
```

这种方式一般用在内 RAM 资源相对紧张而且对效率要求不高的项目中，效率不如上面的方法。如果将上面语句改为如下的形式。

```
uchar xdata tmp[10];
uchar data * xdata pstr;
pstr=tmp;
```

这样方式也是错误的。这次是把 pstr 分配到了外 RAM 区了。C51 语言中一个指针变量最多占用三个字节，对于没有声明指针指向存储空间类型的指针，系统编译代码时都强制加载一个字节的指针类型分辨值。如果将上面改为如下的形式。

```
uchar xdata tmp[10];
uchar * pstr;
pstr=tmp;
```

这是最直接最简单的指针变量声明，但其效率也最低。既把 pstr 分配在外 RAM 空间了又增加了指针类型分辨值。

对于上面得几种访问方式，有的效率高，有的效率低，在实际编程中，要因情况而定。一般来说，51 系列单片机的内部 RAM 资源比较紧张，建议在定义函数内部或程序段内部的局部变量使用内 RAM，而尽量不要把全局变量声明在片内 RAM 中。

！注意：在 C51 语言程序中，如果需要将存储器指针作为一个实参，传递给需要一般指针的函数时，存储器指针将自动转化为一般指针。但如果不说明外部函数原型，基于存储器的指针自动转化为一般指针，有可能导致错误，因而需要用 include 说明所有函数原型。可以强行改变指针类型。

15.6 动态内存分配

动态内存分配用于对一些动态结构进行存储空间的分配。其中这些动态结构可以是树和链表等。对于大多数情况，应尽可能在编译的时候确定所需要的内存空间，并进行分配，例如数组的大小需要明确指定。而对于一些程序，采用动态内存分配可以实现数据的灵活处理。

15.6.1 C51 语言的动态分配函数

在 C51 语言中提供了多个动态内存分配函数。这些函数要求用户声明一个字节数组作为堆栈，根据所需要的动态内存大小来决定数组的长度，被声明的数组存放在 xdata 区中，库函数可以使用特定指针来进行寻址。

C51 提供的动态分配函数示例如下所述。

- **init_mem():** 此函数的功能是初始化动态内存区的程序源代码。可以指定动态内存的位置及大小，只有使用了 init_mem() 才可以调回其他函数，如 malloc、calloc、realloc 等。
- **calloc():** 此函数是给数组分配内存，可以指定单位数据类型及该单元数目。
- **malloc():** 此函数的功能是分配一段固定大小的内存。
- **realloc():** 此函数的功能是调整当前分配动态内存的大小。
- **free():** 此函数的功能是重新分配空间。

在程序中一旦声明了字节数组，指向数组的指针和数组的大小都要传递给初始化函数 init_mem，初始化函数将设置一些内部变量，并进行一些准备工作，用于对动态存储空间进行初始化。初始化工作完成后，便可以在程序中调用后面的各种动态分配函数。

具体函数的使用可以参见前面章节的介绍。

15.6.2 malloc 和 calloc 函数

malloc 和 calloc 函数都是用来动态分配内存空间的。但两者在使用时稍有区别，主要体现在函数形式及内存空间的分配上。malloc 函数和 calloc 函数具有不同的使用形式，下面分别进行讲解。

- **malloc 函数的原型如下：**

```
void *malloc(size_t size);
```

其中包含一个参数 unsigned int，即要分配的内存空间的大小，该函数返回一个指针。

- **calloc 函数的原型如下：**

```
void *calloc(size_t numElements, size_t sizeOfElement);
```

其中，包含两个参数，分别为元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小。

如果调用成功，malloc 和 calloc 函数都将返回所分配的内存空间的首地址。另外，函数 malloc 不能初始化所分配的内存空间，而函数 calloc 则可以。下面分别进行讲解。

- 由于 malloc 函数不能初始化所分配的内存空间，因此如果分配的内存空间原来没有被使用过，则其中的每一位可能都是 0；如果这部分内存曾经被分配过，则其中可能遗留有各种各样的数据。
- 函数 calloc 会将所分配的内存空间中的每一位都初始化为 0，即如果是为字符类型或整数类型的元素分配内存，那么这些元素将保证会被初始化为 0；如果是为指针类型的元素分配内存，那么这些元素通常会被初始化为空指针；如果是为实型数据分配内存，则这些元素会被初始化为浮点型的零。

15.7 小结

本章详细讲述了 C51 语言的存储器结构、存储类型、存储模式及存储器指针等，后面还介绍了动态内存分配。数据的存储模式是单片机系统特有的概念，这里的内容涉及单片机的硬件资源比较多，读者应该对照着单片机的介绍来进行学习，这样可以加深理解。

第四篇 51 系列单片机编程指南篇

第 16 章 51 系列单片机的 指令系统

单片机的指令系统是指单片机 CPU 所能执行的所有指令的集合。其中单片机的指令是执行特定功能的操作命令。单片机使用这些指令，可以实现外部数据信息的读取，内部 CPU 的算法和流程处理，以及对外部接口设备进行控制等。

本章将介绍 51 系列单片机指令系统，首先介绍指令的 7 种寻址方式，即寄存器寻址、寄存器间接寻址、直接寻址、立即寻址、变址寻址、相对寻址和位寻址。对应于每个寻址方式，均给出了在实际单片机程序中的应用示例。接下来着重介绍指令系统中的各类指令，包括数据传送类指令、算术运算类指令、逻辑运算与移位类指令、控制转移类指令、位操作指令和空操作指令，并详细分析各类指令的格式、功能、使用方法及注意事项，并给出了实际的编程示例。

16.1 指令系统简介

单片机的性能主要体现在两个方面，即单片机的硬件资源和单片机的指令系统。硬件资源指的是并行端口的数量、中断的类型和数量、是否自带 A/D 或 D/A、是否带有 I2C 总线等；而单片机的指令系统则主要是单片机 CPU 所能完成的操作，以及指令的执行周期等。

不同类型的单片机，其指令系统一般是不一样的，各个厂商有各自的定义。但为了兼容性的考虑，对于 51 系列的单片机，其指令集基本一致。因此，熟悉了其中一种的指令系统，对 51 系列的任何单片机都可以很容易操作。

16.1.1 指令格式

单片机的指令格式是指令的表示方法。一条指令通常由操作码和操作数两部分组成。其中操作码规定指令执行什么操作，即指令的助记符；而操作数是操作的对象。操作数可以是一个具体的数据，也可以是存储数据的地址或者寄存器。指令的基本格式如下：

操作码	操作数 (地址、寄存器或立即数)
-----	--------------------

其中，操作码和操作数之间至少需要 1 个空格或制表符隔开。操作码是必需的，而操作数有时不是必需的。因为有些指令具有默认的操作对象，因此不需要跟操作数。有的时候一条指令中可以有多个操作数，操作数和操作数之间需要用“,”分隔。

汇编语言编写的程序必须翻译成单片机可执行的机器码。根据机器码的长短，可分为单字节、2 字节和 3 字节等不同长度的指令。这三种指令在单片机的程序存储器中分别占用 1 个、2 个和 3 个存储单元。

1. 单字节指令

单字节指令是指指令的机器码只有一个字节。对于这种单字节指令，操作码和操作数均在

其中，因此在程序存储器中只占用一个字节，其机器码只有一个字节。单字节指令的功能明确专一，操作简单。示例如下：

```
INC DPTR
```

其功能为数据指针加 1，二进制指令码如下：

10100011

另外，有些指令的操作数为工作寄存器 R0～R7，寄存器的编码可用 3 位二进制数表示，这样指令占用 1 个字节也就够了。示例如下：

```
MOV A, Rn
```

这个指令便是单字节指令，对于不同的工作寄存器，单字节指令的机器码如表 16-1 所示。

表 16-1 指令 MOV A, Rn 指令码

指 令	指令码（机器码）	
	二 进 制	十六进制
MOV A, R0	11101000	E8H
MOV A, R1	11101001	E9H
MOV A, R2	11101010	EAH
MOV A, R3	11101011	EBH
MOV A, R4	11101100	ECH
MOV A, R5	11101101	EDH
MOV A, R6	11101110	EEH
MOV A, R7	11101111	EFH

2. 2 字节指令

2 字节指令是指令的机器码需要两个字节来表示。在 2 字节指令的机器码中，第一个字节表示操作码，第二个字节表示操作数。示例如下：

```
MOV A, #data
```

这个指令的功能为将立即数#data 传送到累加器 A 中，指令码如下：

01110100
立即数

例如，对于指令“MOV A, #39H”，其指令码便为 7439H。

3. 3 字节指令

3 字节指令是指令的机器码需要三个字节来表示。在 3 字节指令的机器码中，操作码占一个字节，操作数占两个字节。其中操作数既可以是数据，也可以是地址。示例如下：

```
ANL direct, #data
```

该指令的功能为直接将地址单元 direct 中的内容与立即数#data 进行“与”操作，结果存放于直接地址单元，其指令码如下：

0101 0011
直接地址
立即数

例如，对于指令“ANL 35H,#20H”，其指令码为 533520H。

16.1.2 指令符号

指令符号是为了介绍 51 系列单片机指令系统的方便，而采用的一些符号。这些符号主要用于说明寄存器、地址或数据等。本书所采用的指令符号如下：

- Rn：其中 n=0～7，为当前选中的工作寄存器组中的寄存器 R0～R7 之一；

- **Ri**: 其中 $i=0, 1$, 为当前选中的工作寄存器组中可作为地址指针的寄存器 **R0** 或 **R1**;
 - **#data**: 8 位立即数, 其中 **#** 为立即数的标识符, 例如 **#45H**;
 - **#data16**: 16 位立即数, 例如 **#1234H**;
 - **direct**: 直接寻址符号。指单片机内部 **RAM** 的 8 位地址, 既可以是内部 **RAM** 的低 128 个单元地址, 也可以是特殊功能寄存器的单元地址或符号;
 - **addr11**: 11 位二进制地址码。提供 0~10 共 11 位地址, 而高 5 位地址码不变, 可寻址 2KB 地址空间的任何单元, 只限于在 **ACALL** 和 **AJMP** 指令中使用;
 - **addr16**: 16 位二进制地址。提供 16 位二进制地址, 可寻址 64KB 地址空间的任何单元, 只限于在 **LCALL** 和 **LJMP** 指令中使用;
 - **rel**: 带符号的 8 位二进制码偏移量符号, 一般以二进制补码形式表示, 在相对转移指令中使用;
 - **bit**: 表示直接为寻址的内部 **RAM** 或可位寻址区的特殊功能寄存器的位地址;
 - **@**: 在间接寻址方式中, 表示间址寄存器的前缀标志;
 - **C**: 进位标志位, 或者布尔位处理的累加器, 称之为位累加器;
 - **/**: 一般在位地址的前面, 用于表示对该位先求反再参与操作, 操作完成后不影响该位的值;
 - **(X)**: 表示由 **X** 指定的地址单元或寄存器中的内容;
 - **((x))**: 由 **X** 寄存器的内容作为地址的存储单元的内容;
 - **\$**: 本条指令的起始地址;
 - **←**: 指令操作流程符, 表示将箭头右边的内容送到箭头左边的单元中。
- 这些指令符号在以后的内容介绍及程序设计中经常用到, 读者需要熟悉掌握。

16.2 寻址方式

寻址方式是单片机 **CPU** 在规定的寻址空间能快速获得操作数的方式。通过它, 单片机可以快速、灵活地寻址操作数。操作数是指令系统的一个重要的组成部分, 它指出了指令运算或操作中的数据或数据所在单元的地址。

寻址方式越丰富, 可以为单片机程序设计提供更多方便, 在某些时候可以提高程序的执行速度。但随着指令的增多, 指令系统也就越复杂, 也会给程序的学习和编写带来不便。

本书将讲解 7 种寻址方式, 分别为立即寻址、直接寻址、寄存器寻址、寄存器间接寻址、变址寻址、相对寻址和位寻址。

16.2.1 立即寻址

立即寻址是通过立即数来访问操作数的。其中, 立即数是寻址指令中直接出现的数据。一般来说立即数前面都有标识符 “**#**”, 以便于和直接寻址指令中的直接地址相区别。

立即寻址的立即数可以表示为十六进制、十进制、八进制及二进制, 分别在立即数结尾用字符 **H**、**D**、**O** 和 **B** 来区别。一般情况下, 十进制表示的立即数末尾的字符 **D** 可以省略。立即寻址示例如下:

```
MOV    A,    #21H
```

其中, “**21H**” 就是十六进制立即数, 该指令的功能是把 **21H** 这个数本身送到累加器 **A** 中。指令的操作码为 **74H**, 操作数 **21H**, 如图 16-1 所示。

另外, 立即数可以是 8 位的, 也可以是 16 位的。示例如下:

```
MOV    DPTR,    #1234H
```

其中, “**1234H**” 为 16 位立即数, 该指令将立即数的高 8 位 **12H** 送入 **DPH**, 将低 8 位 **34H** 送入 **DPL**。该指令的操作码是 **90H**, 操作数是 **1234H**, 如图 16-2 所示。

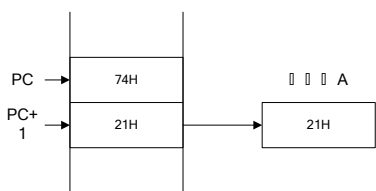


图 16-1 MOV A,#21H 指令执行示意图

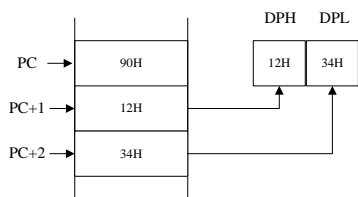


图 16-2 MOV DPTR,#1234H 指令执行示意图

在 51 单片机的指令系统中，只有这条指令的立即数为 16 位，因为 DPTR 为 16 位的寄存器。这条指令用于向地址指针传送 16 位的地址值。

说明：立即寻址方式的寻址空间是程序存储器，也就是说立即寻址方式所操作的立即数是存储在指令字节中的。

下面通过给单片机的并口进行赋值来讲解立即寻址方式在程序中的应用，硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:    MOV     P0, #01H      ;十六进制立即数
          MOV     P1, #100      ;十进制立即数
          MOV     P2, #100B     ;二进制立即数
          MOV     P3, #100      ;八进制立即数
          JMP     START        ;跳转
          END
```

这里采用立即寻址方式将单片机的 P0、P1、P2 和 P3 端口分别输出具体的数值。在程序中，立即寻址数的数值分别采用十六进制、十进制、二进制和八进制进行表示。

将程序在 Keil μ Vision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0、P1、P2 和 P3 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μ Vision3 中进行单片机端口仿真，仿真结果如图 16-3 所示。

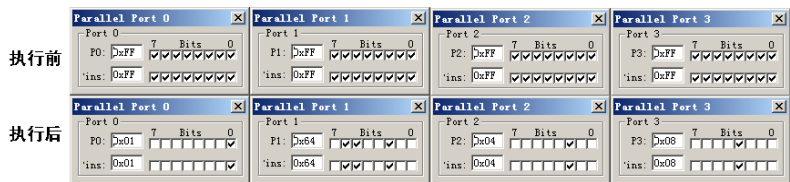


图 16-3 端口仿真结果

16.2.2 直接寻址

直接寻址方式是在指令中直接给出操作数的单元地址的寻址方式。此时，指令的操作数部分是操作数所存放的物理地址，而不是实际的数值。


利用直接寻址方式，可寻址如下两类存储空间。

- 单片机片内 RAM 低 128 个字节单元，在指令中直接地址以单元地址形式给出。单片机片内低 128 字节的地址为 00H~7FH。这样，在这 128 位地址的直接寻址方式中，00H 代表第 0 位，01H 代表第 1 位，依次类推；
- 特殊功能寄存器。对于特殊功能寄存器，其直接地址还可以用特殊功能寄存器的符号名称来表示。

典型的直接寻址指令示例如下：

```
MOV      A,    3DH
```

其中，“3DH”就是表示直接地址，该指令的功能是把内部 RAM 地址为 3DH 中的内容 2FH 传给累加器 A。指令的操作码是 E5H，示意图如图 16-4 所示。

说明：在单片机中，直接寻址是访问特殊功能寄存器的唯一方法。

另外，要特别注意立即数和直接地址的区别，示例如下：

```
MOV 3DH, #3DH
```

该指令中第一个 3DH 为直接地址，采用的是直接寻址方式；第二个#3DH 为立即数，采用的是立即寻址。该指令的操作码是 75H，操作示意如图 16-5 所示。

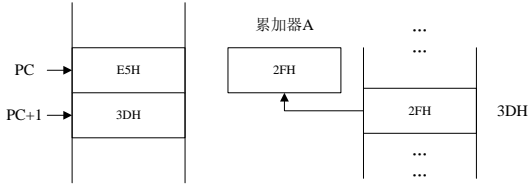


图 16-4 MOV A,3DH 指令执行示意图

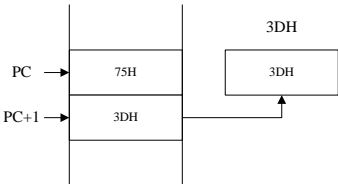


图 16-5 MOV 3DH,#3DH 指令执行示意图

下面通过给单片机的并口进行赋值来讲解直接寻址方式在程序中的应用，硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:   MOV      34H, #00H      ;立即寻址方式赋值
        MOV      35H, #01H      ;立即寻址方式赋值
        MOV      36H, #02H      ;立即寻址方式赋值
        MOV      37H, #03H      ;立即寻址方式赋值
        MOV      P0, 34H        ;直接寻址方式写端口 P0
        MOV      P1, 35H        ;直接寻址方式写端口 P1
        MOV      P2, 36H        ;直接寻址方式写端口 P2
        MOV      P3, 37H        ;直接寻址方式写端口 P3
        JMP      START          ;跳转
END
```

在该程序中，首先采用立即寻址方式将单片机片内 RAM 的 34H、35H、36H 和 37H 地址处进行赋值；然后使用直接寻址方式将 34H、35H、36H 和 37H 地址处的数值直接输出到单片机的 P0、P1、P2 和 P3 端口上。

将程序在 Keil μVision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0、P1、P2 和 P3 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μVision3 中进行单片机端口仿真，仿真结果如图 16-6 所示。

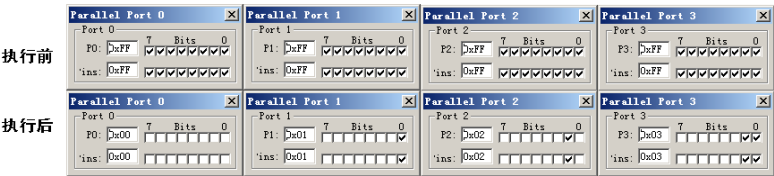


图 16-6 端口仿真结果

16.2.3 寄存器寻址

寄存器寻址就是在指令的操作数位置上指定寄存器的寻址方式。其中，寄存器的内容作为操作数。采用寄存器寻址方式的指令一般都是一个字节的指令。这种方式的可以寻址寄存器

R0~R7、A 和 AB，以及数据指针 DPTR。示例如下：

```
MOV    A,    R1
```

该指令是将寄存器 R1 中的内容传送到累加器 A 中。指令的操作码是 E9，可参阅表 16-1，指令执行的示意图，如图 16-7 所示。

寄存器在单片机 CPU 的内部，因此采用寄存器寻址的速度相比其他几种寻址方式要快，可以使程序具有较高的运算处理速度。

下面通过给单片机的并口进行赋值来讲解寄存器寻址方式在程序中的应用，硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      R0,#00H      ;立即寻址方式赋值
          MOV      R1,#11H      ;立即寻址方式赋值
          MOV      R2,#22H      ;立即寻址方式赋值
          MOV      R3,#33H      ;立即寻址方式赋值
          MOV      P0,R0         ;寄存器 R0 寻址方式写端口 P0
          MOV      P1,R1         ;寄存器 R1 寻址方式写端口 P1
          MOV      P2,R2         ;寄存器 R2 寻址方式写端口 P2
          MOV      P3,R3         ;寄存器 R3 寻址方式写端口 P3
          JMP      START        ;跳转
END
```

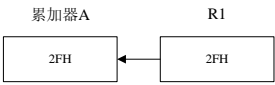


图 16-7 MOV A,R1 指令执行示意图

在该程序中，首先使用立即寻址方式为寄存器 R0、R1、R2 和 R3 赋值。然后使用寄存器寻址方式将寄存器 R0、R1、R2 和 R3 中的数值直接输出到单片机的 P0、P1、P2 和 P3 端口上。

将程序在 Keil μVision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0、P1、P2 和 P3 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μVision3 中进行单片机端口仿真，仿真结果如图 16-8 所示。

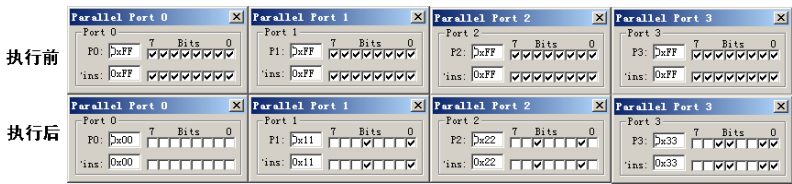


图 16-8 端口仿真结果

16.2.4 寄存器间接寻址

寄存器间接寻址是将操作数保存在 RAM 中，而该 RAM 的地址放在寄存器中，通过访问寄存器来获得 RAM 中操作数的寻址方式。

寄存器间接寻址可以访问如下所述的存储空间。

- 内部 RAM 低 128 个字节单元，可以采用 R0、R1 作为间接寻址的寄存器；
- 外部 RAM。这里有两种形式：一是采用 R0、R1 作为间接寻址的寄存器，可寻址 256 个单元；二是采用 16 位的 DPTR 作为间接寻址的寄存器，可寻址外部 RAM 的完整 64KB 地址空间。

寄存器间接寻址也用到了寄存器，为了区别寄存器寻址和寄存器间接寻址，在寄存器名称前加“@”标识符，来表示寄存器间接寻址。示例如下：

```
MOV     A,    @R1
```

51 单片机开发与应用技术详解

这时寄存器 R1 的内容 3DH 是操作数在 RAM 中的地址，内部 RAM 的 3DH 单元的内容 2FH 才是操作数。该指令把该操作数传送到累加器 A，结果累加器 A 中的内容变为 65H。该指令的操作码是 E7，指令执行的示意图，如图 16-9 所示。

若不加 “@” 标识符，则是寄存器寻址指令。示例如下：

```
MOV    A,    R1
```

上面语句执行结果累加器 A 中的内容为 3DH。寄存器寻址和寄存器间接寻址很容易混淆，这里应该区分清楚，其操作的内容是不一样的。

另外，寄存器间接寻址也可以用于目的操作数，示例如下：

```
MOV  @R1,A
```

该指令中，寄存器 R1 中存有 RAM 地址 3DH，指令将累加器 A 中的内容送入 RAM 的 3DH 单元。该指令的操作码是 F7，指令执行的示意图，如图 16-10 所示。

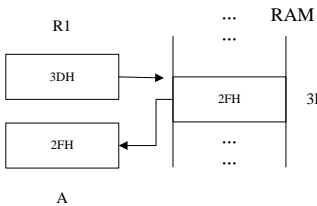


图 16-9 MOV A,@R1 指令执行示意图

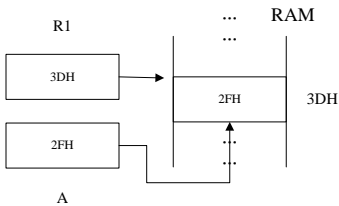



图 16-10 MOV @R1,A 指令执行示意图

说明：在 51 系列单片机中，只能使用寄存器 R0、R1 和数据指针 DPTR 作为间接寻址的寄存器。

下面通过给单片机的并口进行赋值来讲解寄存器间接寻址方式在程序中的应用，硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      34H,#00H      ;立即寻址方式赋值
          MOV      35H,#01H      ;立即寻址方式赋值
          MOV      36H,#02H      ;立即寻址方式赋值
          MOV      37H,#03H      ;立即寻址方式赋值
          MOV      R0,#34H        ;立即寻址方式赋值
          MOV      R1,#35H        ;立即寻址方式赋值
          MOV      P0,@R0         ;寄存器间接寻址方式写端口 P0
          MOV      P1,@R1         ;寄存器间接寻址方式写端口 P1
          MOV      R0,#36H        ;立即寻址方式赋值
          MOV      R1,#37H        ;立即寻址方式赋值
          MOV      P2,@R0         ;寄存器间接寻址方式写端口 P2
          MOV      P3,@R1         ;寄存器间接寻址方式写端口 P3
          JMP      START          ;跳转
          END
```

在该程序中，首先使用立即寻址方式为片内 RAM 的 34H、35H、36H 和 37H 单元赋值，然后分别将 RAM 的地址保存到寄存器 R0 和 R1 中。最后，使用寄存器间接寻址方式分别将寄存器 R0 和 R1 中的数值直接输出到单片机的 P0、P1、P2 和 P3 端口上。

将程序在 Keil μVision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0、P1、P2 和 P3 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μVision3

中进行单片机端口仿真，仿真结果如图 16-11 所示。

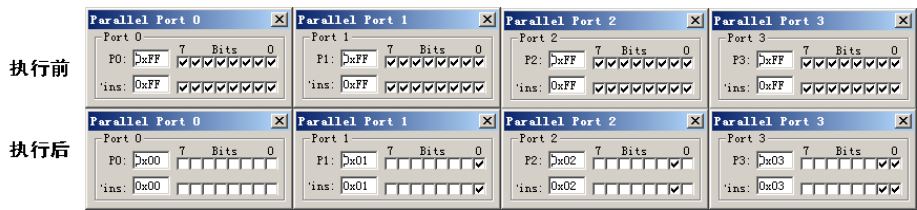


图 16-11 端口仿真结果

16.2.5 变址寻址

变址寻址是以基址寄存器加变址寄存器所形成的 16 位地址作为操作数地址的寻址方式。其中基址寄存器为 DPTR 或 PC，而累加器 A 存放地址偏移量作为变址寄存器。变址寻址方式又称为基址寄存器+变址寄存器的间接寻址。

相对寻址方式的指令都是单字节指令，一般常用于访问程序存储器中的数据表格。示例如下：

```
MOV    A,    @A+DPTR
MOV    A,    @A+PC
```

第一条指令的功能是将累加器 A 的内容与数据指针 DPTR 的内容相加形成操作数的地址，即程序存储器的 16 位地址，再取该地址中的内容送入累加器 A 中。该指令的操作码为 93H，指令的执行示意图，如图 16-12 所示。

第二条指令的功能是将 A 的内容与 PC 的内容相加形成操作数的地址，即 ROM 16 位地址，把该地址中的内容送入累加器 A 中。该指令的操作码为 83H。

下面通过给单片机的并口进行赋值来讲解变址寻址方式在程序中的应用，硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG    0000H
JMP    START
ORG    1000H
START:  MOV    A,    #09H
        MOV    DPTR, #1000H
        JMP    @A+DPTR      ;变址寻址，跳转
        MOV    P0, #11H      ;3 字节指令，跳过，执行不到
        MOV    P1, #22H      ;P1 端口输出
        MOV    A,    #04H
        MOVC A,    @A+PC      ;变址寻址，指向表的第一个数
        MOV    P2, A          ;P2 端口输出
        JMP    START          ;跳转
BIAO:   DB    01H, 02H, 03H, 04H ;数据表
        DB    11H, 12H, 13H, 14H
        DB    21H, 22H, 23H, 24H
END
```

在该程序中，首先使用立即寻址方式为累加器 A 和数据指针 DPTR 赋值，然后通过变址寻址指令 JMP 跳转指令跳转到 P1 端口的输出行，这里跳过了 P0 端口的输出行。接着，同样采用立即寻址方式为累加器 A 赋值，并通过程序存储器 PC 来寻址数据表 BIAO 中的数据并在 P2 端口赋值，这里变址寻址指向数据表的第一个数。

将程序在 Keil μVision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0、P1 和 P2 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μVision3 中进行单片机端口仿真，仿真结果如图 16-13 所示。

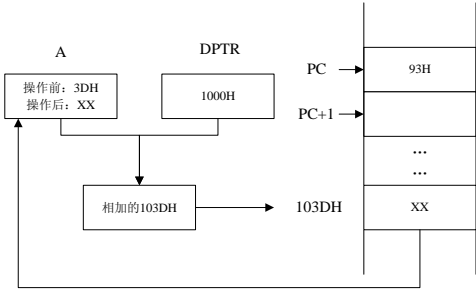


图 16-12 MOV A,@A+DPTR 示意图

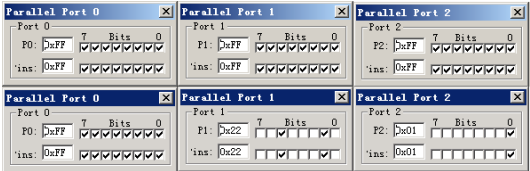


图 16-13 端口仿真结果

16.2.6 相对寻址

相对寻址是由 PC 值和相对地址偏移量 `rel` 共同构成地址的寻址方式。其中相对地址偏移量 `rel` 在指令中由操作数给出。程序中将程序计数器 PC 的当前值与指令中给出的偏移量 `rel` 相加，其结果作为转移地址送入 PC 中，即跳向一个新的地址来执行程序。相对寻址是在相对转移指令中，修改 PC 指针的值，故可用来实现程序的分支转移。

地址偏移量 `rel` 是一个带符号的 8 位二进制数，其取值范围为 `-128~+127`，故 `rel` 给出的跳转范围是当前 PC 的 `-128~127` 之间的某段程序存储器地址中。

注意：PC 指针的当前值，是指正在执行指令的下条指令的地址，而不是当前指令的地址。

相对寻址指令示例如下：

```
SJMP 33H
```

这是无条件相对转移指令，该指令占用两个字节，因此 PC 指针的当前值为 `PC+2`。地址偏移量 `rel=33H`，则转移地址为 `PC+2+33H`，程序则转向 `PC+2+33H` 的位置继续执行程序。该指令的操作码为 `80H`，指令执行的示意图，如图 16-14 所示。

这里应该注意，实际在 Keil μ Vision3 中进行程序设计的时候，一般不直接写地址，而是直接写程序段的标号，因为地址计算比较复杂。因此 Keil μ Vision3 编译的时候，将其翻译成二进制的机器语言，再进行相对寻址的跳转。示例程序如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:   MOV     A, #10H      ;立即寻址赋值
         MOV     R0, #20H     ;立即寻址赋值
         MOV     R1, #30H     ;立即寻址赋值
         SJMP    1000H        ;跳转
END
```

在该程序中，使用立即寻址完成赋值操作，最后使用相对寻址的跳转指令，跳转到 `1000H` 处循环执行。在 Keil μ Vision3 中进行编译，查看其机器码，如图 16-15 所示。

`SJMP` 指令地址跳转的目的地址=`PC 当前值+rel=0x1008+(-8)=0x1000`，即程序段中 `START` 处。在机器码中，`F8` 为“-8”的补码形式。

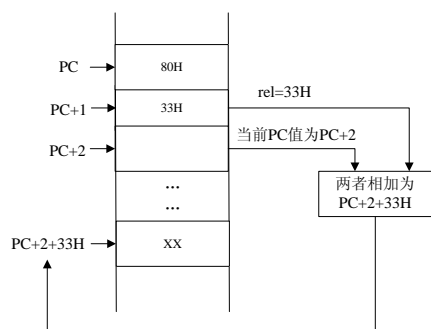


图 16-14 SJMP 33H 指令执行示意图

4: START: MOV	A, #10H	;立即寻址赋值
C:0x1000	7410	MOV A, #0x10
5:		
C:0x1002	7820	MOV R0, #20H ;立即寻址赋值
6:		
C:0x1004	7930	MOV R1, #30H ;立即寻址赋值
7:		
C:0x1006	80F8	SJMP 1000H ;跳转
C:0x1008	00	NOP
C:0x1009	00	NOP

图 16-15 程序的机器码

16.2.7 位寻址

位寻址是指在指令的操作数位置上直接给出位地址的寻址方式。51 系列单片机有位处理功能，可对寻址的位进行操作。在 51 系列单片机中，可以位寻址的范围有如下两类。

- 单片机 RAM 的 20H~2FH 单元，是可位寻址区域，共 16×8=128 位；
- 字节地址能被 8 整除的特殊功能寄存器的相应位。

位寻址指令示例如下：

```
MOV C, 2AH.3
```

该指令的功能是把地址 2AH 中的第 4 位 D3，即 2AH.3 的值（0 或 1）传送到位累加器 CY 中。该指令执行的示意图，如图 16-16 所示。

在 51 系列单片机的指令系统中，位地址的表示可以采用以下几种方式。

- 直接使用位地址。单片机内部 RAM 的可位寻址区 20H~2FH，共 16 个单元，16×8=128 位，其位地址为 00H~7FH，例如，20H 单元的 0~7 位的位地址为 00H~07H。
- 用 RAM 地址加位序号表示。如 2AH.3 表示 2AH 单元的 D3 位。
- 对于特殊功能寄存器，其中的可寻址位均有位名称，可以用位名称来表示该位，如可用 RS0 表示 PSW 中的 D2 位（D30H.2）。
- 对特殊功能寄存器也可以直接用寄存器符号加位序号表示。如 PSW 中的 D3 位，可以表示为 PSW.3。

下面分别采用上述的这几种方式来讲解变址寻址方式在程序中的应用，硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG 0000H
JMP START
ORG 1000H

START: SETB 20H.0 ;置 20H.0 为 1
      SETB 01H ;置 20H.1 为 1
      SETB 03H ;置 20H.3 为 1
      MOV P0, 20H ;将片内 RAM 的 20H 单元数据输出到 P0 端口
      MOV P1, 00H ;P1 端口清零
      SETB P1.0 ;置 P1.0 为 1
      SETB RS0 ;特殊功能寄存器 PSW 的 D2 位
      SETB PSW.3 ;特殊功能寄存器 PSW 的 D3 位
      JMP START ;跳转
      END
```

在该程序中，首先为片内 RAM 的 20H 单元采用位寻址方式赋值，然后将 20H 单元的数据输出到 P0 端口。接着对 P1.0 端口进行位寻址赋值。最后，采用两种方法对特殊功能寄存器的

51 单片机开发与应用技术详解

D2 和 D3 位赋值。

将程序在 Keil μ Vision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0 和 P1 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μ Vision3 中进行单片机端口仿真，仿真结果如图 16-17 所示。

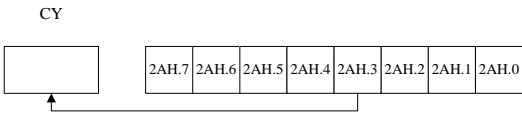


图 16-16 MOV C,2AH.3 指令执行示意图

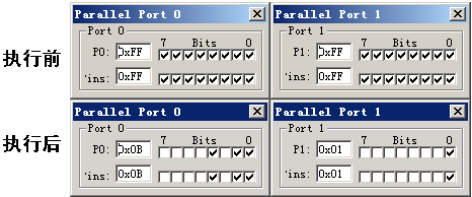


图 16-17 端口仿真结果

16.3 数据传送指令

51 系列单片机的指令系统共有 111 条指令，按功能可以分为如下 5 类。

- 数据传送类指令（29 条）；
- 算术运算类指令（24 条）；
- 逻辑运算及移位指令（24 条）；
- 控制转移指令（17 条）；
- 位操作指令（17 条）。

从这一节开始，将分别介绍以上各种指令。首先是数据传送指令。数据传送指令是把指令的源操作数传送到目的操作数中的指令。数据传送指令是最常用的一类基本指令，主要用于数据的传送和保存，以及交换数据等。

数据传送指令执行后，目的操作数被源操作数所替代，而源操作数保持不变。51 系列单片机指令系统中的数据传送指令共有 29 条，下面分别详细介绍这些指令。

16.3.1 内部 RAM 数据传送指令

内部 RAM 数据传送指令指数据的传送是在单片机的片内 RAM 中进行的。51 系列单片机指令系统中，共有 16 条与内部 RAM 数据传送有关的指令。按照目的操作数的不同，内部 RAM 数据传送指令，可以分为 5 类，分别介绍如下：

1. 以累加器 A 为目的操作数的数据传送指令

以累加器 A 为目的操作数的数据传送指令，是将源操作数中的内容传送到累加器 A 中。源操作数可以采用以下 4 种寻址方式。

- 立即寻址。示例如下：

```
MOV    A,    #5F                ; A←5F
```

- 直接寻址。示例如下：

```
MOV    A,    4AH                ; A←( 4A )
```

- 寄存器寻址。示例如下：

```
MOV    A,    R1                ; A←( R1 )
```

- 寄存器间接寻址。示例如下：

```
MOV    A,    @R1               ; A←( ( R1 ) )
```

下面分别采用上述的这几种方式，来讲解以累加器 A 为目的操作数的数据传送指令在程序中的应用。硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG    0000H
```

```
JMP      START
ORG      1000H
START:   MOV      4AH, #4AH      ; (4A) ← 4A
        MOV      R0, #01H      ; R0 ← 01
        MOV      40H, #40H     ; (40) ← 40
        MOV      R1, 40H       ; R1 ← 40
        MOV      A, #5FH       ; 立即寻址 A ← 5F
        MOV      A, 4AH        ; 直接寻址 A ← ( 4A )
        MOV      A, R0         ; 寄存器寻址 A ← ( R0 )
        MOV      A, @R1        ; 寄存器间接寻址 A ← ( ( R1 ) )
        JMP      START        ; 跳转
END
```

在该程序中，首先为 RAM 的 4AH 单元，以及寄存器 R0 和 R1 赋值；然后分别采用立即寻址、直接寻址、寄存器寻址和寄存器间接寻址的方式来向累加器 A 进行数据传送。

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在每一步数据传送指令执行时，累加器 A 中数值的变化。

2. 以直接地址为目的的操作数的数据传送指令

以直接地址为目的的操作数的数据传送指令，是将源操作数中的内容传送到由直接地址 direct 所指定的片内 RAM 存储单元中。源操作数可以采用以下 4 种寻址方式。

➤ 立即寻址。假设目的地址为 37H，示例如下：

```
MOV      37H, #2DH            ; 37H ← 2D
```

➤ 直接寻址。假设目的地址为 37H，源地址为 6FH，示例如下：

```
MOV      37H, 6FH             ; 37H ← ( 6FH )
```

！注意：“MOV direct1, direct2”指令在译成机器码时，源地址在前，目的地址在后。

➤ 寄存器寻址。假设目的地址为 37H，示例如下：

```
MOV      37H, Rn               ; 37H ← ( Rn )
```

```
MOV      37H, A                ; 37H ← ( A )
```

➤ 寄存器间接寻址。假设目的地址为 37H，示例如下：

```
MOV      37H, @Ri              ; 37H ← ( ( Ri ) )
```

下面分别采用上述的这几种方式，来讲解以直接地址为目的的操作数的数据传送指令在程序中的应用。硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:   MOV      A, #10H
        MOV      R0, #0ABH
        MOV      R1, #39H
        MOV      37H, #2DH     ; 立即寻址
        MOV      38H, 37H     ; 直接寻址
        MOV      39H, R0      ; 寄存器寻址
        MOV      40H, A       ; 寄存器寻址
        MOV      41H, @R1     ; 寄存器间接寻址
        JMP      START        ; 跳转
END
```

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在每一步数据传送指令执行时，单片机片内 RAM 中数据的变化。仿真结果如图 16-18 所示。

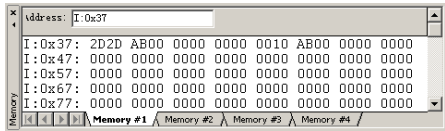


图 16-18 RAM 数据仿真

3. 以寄存器 Rn 为目的操作数的数据传送指令

以寄存器 Rn 为目的操作数的数据传送指令，是将源操作数所指定的内容传送到当前工作寄存器组 R0~R7 中的某个寄存器中。源操作数可以采用以下三种寻址方式。

➤ 立即寻址。示例如下：

```
MOV      Rn, #8DH                ;Rn←8DH
```

➤ 直接寻址。示例如下：

```
MOV      Rn, 67H                 ;Rn←( 67H )
```

➤ 寄存器寻址。示例如下：

```
MOV      Rn, A                   ;Rn←( A )
```

！注意：以下两个指令在 51 系列单片机指令系统中是没有意义的。

```
MOV      Rn, Rn
MOV      Rn, @Ri
```

下面分别采用上述的这几种方式，来讲解以寄存器 Rn 为目的操作数的数据传送指令在程序中的应用。硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:   MOV      A, #10H
MOV      67H, #43H
MOV      R0, #8DH                ;立即寻址
MOV      R1, 67H                 ;直接寻址
MOV      R2, A                   ;寄存器寻址
JMP      START                  ;跳转
END
```

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以看到数据传送指令执行后，寄存器 R0、R1、R2 中数值分别变为 0x8DH、0x43H 和 0x10H。

4. 以间接地址 @Ri 为目的操作数的数据传送指令

以间接地址 @Ri 为目的操作数的数据传送指令，是将源操作数所指定的内容传送到以 R0 或 R1 为地址指针的片内 RAM 单元中。源操作数可以采用以下三种寻址方式。

➤ 立即寻址。示例如下：

```
MOV      @Ri, #43H               ;( Ri )←43H
```

➤ 直接寻址。示例如下：

```
MOV      @Ri, 51H               ;( Ri )←( 51H )
```

➤ 寄存器寻址。示例如下：

```
MOV      @Ri, A                 ;( Ri )←( A )
```

！注意：在 51 系列单片机指令系统中，如下的指令是没有意义的。

```
MOV @Ri,Rn
```

下面分别采用上述的这几种方式，来讲解以间接地址@Ri 为目的操作数的数据传送指令在程序中的应用。硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      R0,#10H
          MOV      @R0,#43H      ;立即寻址
          MOV      R1,#11H
          MOV      51H,#7AH
          MOV      @R1,51H      ;直接寻址
          MOV      R0,#12H
          MOV      52H,#55H
          MOV      A,52H
          MOV      @R0,A        ;寄存器寻址
          JMP      START        ;跳转
END
```

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以看到数据传送指令执行后，RAM 存储单元 10H、11H、12H 中数值分别变为 0x43H、0x7AH 和 0x55H，如图 16-19 所示。

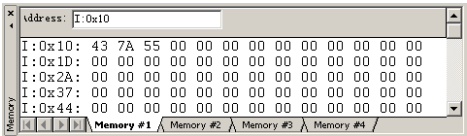


图 16-19 程序仿真结果

5. 以 DPTR 为目的操作数的数据传送指令

以 DPTR 为目的操作数的数据传送指令是将外部存储器（RAM 或 ROM）某单元地址作为立即数传送到 DPTR 中。其中源操作立即数为 16 位常数，这是 51 系列单片机指令系统中唯一的一条 16 位立即数传送指令。

由于 16 位的数据指针 DPTR 由 DPH 和 DPL 组成，因此，指令执行的结果是将立即数的高 8 位送到 DPH 中，低 8 位送到 DPL 中。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      DPTR,#5678H  ;DPTR←5678H
          JMP      START        ;跳转
END
```

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以看到数据传送指令执行后，数据指针 DPTR 的数值变为 0x5678H。

6. 需要注意的事项

首先，对于汇编语言程序设计，有时需要注意指令的字节数。对于这里所介绍的这些指令，有以下 4 种情况。

- 如果既不包含立即数，也不包含直接寻址的地址，则该指令为单字节；
- 若指令中包含直接地址或 8 位立即数，则该指令为双字节；
- 如果保护两个直接地址或 8 位立即数，则该指令为三字节；
- 以 DPTR 为目的操作数的数据传送指令，为 4 字节指令。

示例如下：

```
MOV A,@R0                ;1 字节
MOV A,direct              ;2 字节
MOV direct,#data          ;3 字节
MOV DPTR,#data16          ;4 字节
```

另外，涉及累加器 A 的指令将影响奇偶标志位 P，而其余的指令则不影响标志位 CY、AC 和 OV。

16.3.2 外部 RAM 数据传送指令

外部 RAM 的数据传送指令是指单片机 CPU 与外部 RAM 存储器进行数据传送的指令。这类指令通过累加器 A 来传送，而且必须采用寄存器间接寻址的方式来获取外部 RAM 的数据。外部 RAM 数据传送指令共有 4 条，按照间接寻址寄存器的不同，可以分为以下两类。

1. 以 DPTR 作为间接寻址寄存器

以 DPTR 作为间接寻址寄存器的外部 RAM 数据传送指令，是在 DPTR 所指定的外部 RAM 单元与累加器 A 之间传送数据。示例如下：

```
MOVX A,@DPTR              ;A←( DPTR )
MOVX @DPTR,A              ;( DPTR )←A
```

由于 DPTR 是 16 位地址指针，通过这两条指令可以寻址片外 RAM 64KB 全部存储空间。

下面分别采用上述的两条指令，来讲解以 DPTR 作为间接寻址寄存器的外部 RAM 数据传送指令在程序中的应用。硬件采用前面介绍的 AT89S52 的最小系统，外扩一定容量的 RAM 存储器芯片即可。程序示例如下：

```
ORG 0000H
JMP START
ORG 1000H
START: MOV DPTR,#1000H      ;源数据地址送 DPTR
      MOVX A,@DPTR         ;从外部 RAM 中取数据送 A
      MOV 4DH,A            ;将累加器 A 中的数据送入 4DH 单元
      MOV A,#34H           ;数据 34H 送入累加器 A
      MOV DPTR,#1001H      ;源数据地址送 DPTR
      MOVX @DPTR,A         ;累加器 A 中的数据送到外部 RAM
      JMP START            ;跳转
END
```

在该段程序，首先将片外 RAM 的 1000H 单元的内容传送到片内 RAM 的 4DH 单元中，然后将数据 34H 送入外部 RAM 的 1001H 单元中。

将程序在 Keil μVision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。当然也可以在 Keil μVision3 中进行单片机仿真。

2. 以 R0 或 R1 作为间接寻址寄存器

以 R0 或 R1 作为间接寻址寄存器的外部 RAM 数据传送指令，是在寄存器 R0 或 R1 所指定的外部 RAM 单元与累加器 A 之间传送数据。示例如下：

```
MOVX A,@Ri                ; A←( Ri )
MOVX @Ri,A                ;( Ri )←A
```

由于 R0 或 R1 只能存储 8 位的数据，因此这两条指令只能寻址 256 个字节外部 RAM 存储单元空间。

下面分别采用上述的两条指令，来讲解以 R0 或 R1 作为间接寻址寄存器的外部 RAM 数据传送指令在程序中的应用。硬件采用前面介绍的 AT89S52 的最小系统，外扩一定容量的 RAM

存储器芯片即可。程序示例如下：

```

ORG      0000H
JMP      START
ORG      1000H

START:    MOV  R0, #10H          ;源数据地址送 R0
          MOVX A, @R0           ;从外部 RAM 中取数据送 A
          MOV  4DH, A           ;将累加器 A 中的数据送入 4DH 单元
          MOV  A, #34H          ;数据 34H 送入累加器 A
          MOV  R1, #11H        ;源数据地址送 R1
          MOVX @R1, A           ;累加器 A 中的数据送到外部 RAM
          JMP  START           ;跳转
END
```

这段程序中，首先将片外 RAM 的 10H 单元的内容传送到片内 RAM 的 4DH 单元中，然后将数据 34H 送入外部 RAM 的 11H 单元中。

将程序在 Keil μ Vision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。当然也可以在 Keil μ Vision3 中进行单片机仿真。

16.3.3 程序存储器数据传送指令

程序存储器数据传送指令是指从程序存储器 ROM 中读取源操作数，并送入累加器 A 中。有两条指令，分别如下：

```

MOVC A, @A+PC          ; A ← ( ( A ) + ( PC ) )
MOVC A, @A+DPTR         ; A ← ( ( A ) + ( DPTR ) )
```

在这两条指令中，源操作数都是采用变址寻址方式，并且都是单字节指令。该指令常用于查阅建立在 ROM 中的数据表格，因此又称为“查表指令”。

这两条指令的功能完全相同，但是使用有一些不同。

1. MOVC A, @A+PC 指令

这条指令是采用 PC 指针作为变址寻址方式的基址寄存器。指针 PC 中的内容为下一条指令的地址，在读取数据表格时，不是表格首地址。这样基址和实际读取的数据表格首地址不一致，使得 A+PC 与实际要访问的单元地址不一致。

因此，在使用该指令进行查表之前，必须先用加法指令对累加器 A 的内容进行调整，确保 A+PC 与所读取的 ROM 单元地址保持一致。由于累加器 A 的内容为 8 位无符号数，取值范围为 0~255B，因此该指令只能查找所在地址 256B 范围内的数据表。为此，常称这条指令也称为“近程查表指令”。

下面采用程序指针 PC 作为基址寄存器，来讲解指令在程序中的应用，示例如下：

```

ORG      0000H
JMP      START
ORG      1000H

START:    MOV  A, #00H          ;查表的元素编号，这里查找第一个数据
          ADD  A, #03H          ; ( A ) + 03H 作为地址调整，该指令占用 2 个字节
          MOVC A, @A+PC         ; ( A ) + data+PC 确定地址，取数送入 A，该指令占用 1 个字节
          MOV  20H, A           ;存放结果，该指令占用 2 个字节
          RET                   ;该指令占用 1 个字节

BIAO:    DB    11H, 12H, 13H, 14H ;数据表
          DB    21H, 22H, 23H, 24H
          DB    31H, 32H, 33H, 34H
END
```

用 PC 作为基址寄存器，PC 指向的是下一条指令的地址，一般不能随意更改。这里需要调整累加器 A 中的数值，来使其指向表格。因此，在该程序中，首先指明要查找的表中的元素，这里查找第一个元素，所以相对地址为 0x00；然后使用 ADD 指令将地址偏移量加到累加器 A 中；最后，使用 MOVC A,@A+PC 指令将表中的数据保存到累加器 A 中，并将其写入片内 RAM 的 20H 单元。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以看到查表指令执行后，RAM 存储单元 20H 中数值分别变为 0x11H，如图 16-20 所示。

2. MOVC A,@A+DPTR 指令

这条指令是采用数据指令 DPTR 作为变址寻址方式的基址寄存器。在进行数据查表前，需要先把 16 位表格首地址传送到指针 DPTR 中。16 位的 DPTR 可以寻址 64KB 的存储范围，这样数据表格可以存放在 64KB ROM 空间的任意位置，使用比较方便，因此该条指令也称为“远程查表指令”。

下面采用数据指针 DPTR 作为基址寄存器，来讲解指令在程序中的应用，示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:    MOV      A, #04H          ;查表的元素编号，这里查找第 5 个数据
          MOV      DPTR, #2000H    ;表的首地址
          MOVC     A, @A+DPTR      ;( A )+DPTR 确定地址，取数送入 A，该指令占用 1 个字节
          MOV      20H, A          ;存放结果，该指令占用 2 个字节
          RET                    ;该指令占用 1 个字节
          ORG      2000H

BIAO:     DB       11H, 12H, 13H, 14H ;数据表
          DB       21H, 22H, 23H, 24H
          DB       31H, 32H, 33H, 34H
          END
```

在该程序中，首先指明要查找的表中的元素，这里查找第 5 个元素，所以相对地址为 0x04。然后将数据表的首地址 2000H 送入数据指针 DPTR。最后，使用 MOVC A,@A+DPTR 指令将表中的数据保存到累加器 A 中，并将其写入片内 RAM 的 20H 单元。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以看到查表指令执行后，RAM 存储单元 20H 中数值分别变为 0x21H，如图 16-21 所示。

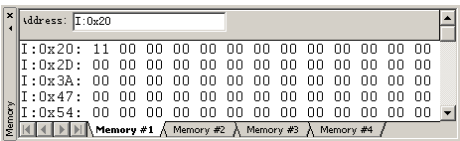


图 16-20 程序仿真结果

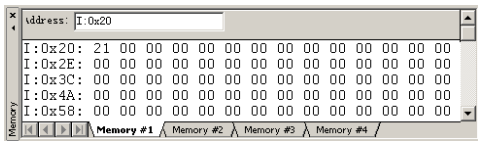


图 16-21 程序仿真结果

16.3.4 数据交换指令

数据交换指令是指在累加器 A 与内部 RAM 单元之间的数据交换指令，这类指令共有 5 个。按照交换的数据可以分为三种，下面分别介绍。

1. 整字节交换指令

整字节交换指令是在累加器 A 与内部 RAM 单元之间进行整字节交换的指令。按照寻址方式的不同，可以有以下三条指令。

➤ 直接寻址。示例如下，指令执行完毕后，A 和地址 direct 中的数据互换。

```
XCH A, direct          ; ( A ) ↔ ( direct )
```


➤ 寄存器寻址。示例如下，指令执行完毕后，A 和寄存器 Rn 中的数据互换。

```
XCH A,Rn ; ( A ) ↔ ( Rn )
```

➤ 寄存器间接寻址。示例如下，指令执行完毕后，A 和寄存器 Ri 所指向的单元中的数据进行互换。

```
XCH A,@Ri ; ( A ) ↔ ( ( Ri ) )
```

2. 半字节交换指令

半字节交换指令是在累加器 A 与寄存器 Ri 间接寻址单元的数据进行半字节交换的指令。

半字节交换指令只有一个，示例如下：

```
XCHD A,@Ri ; ( A ) 低 4 位 ↔ ( ( Ri ) ) 低 4 位
```

指令执行完毕后，将累加器 A 的低 4 位二进制数和 Ri 间接寻址单元的低 4 位二进制数交换，而各自的高 4 位数据保持不变。

3. 累加器高低半字节交换指令

累加器高低半字节交换指令是将累加器 A 的高 4 位二进制数和低 4 位二进制数进行互换。

示例如下：

```
SWAP A ; ( A ) 高 4 位 ↔ ( A ) 低 4 位
```

该指令可以实现十六进制数据或 BCD 码的数位交换。

下面分别采用上述几种数据交换指令来讲解其在程序设计中的应用。程序示例如下：

```
ORG 0000H
JMP START
ORG 1000H
START: MOV A,#04H ;累加器 A 赋值
MOV 20H,#20H ;RAM 的 20H 单元赋值
XCH A,20H ;直接寻址整字节交换指令

MOV R2,#05H ;寄存器 R2 赋值
XCH A,R2 ;寄存器寻址整字节交换指令
MOV 21H,A ;保存数据 05H 到 RAM 的 21H 单元

MOV 10H,#06H ;RAM 单元赋值
MOV R0,#10H ;寄存器 R0 赋值
XCH A,@R0 ;寄存器间接寻址整字节交换指令
MOV 22H,A ;保存数据 06H 到 RAM 的 22H 单元

MOV A,#00H ;累加器 A 赋值
MOV 11H,#0FH ;RAM 单元赋值
MOV R1,#11H ;寄存器 R1 赋值
XCHD A,@R1 ;半字节交换指令
MOV 23H,A ;保存数据到 RAM 的 23H 单元，数据应该为 0FH

MOV A,#0FH ;累加器 A 赋值
SWAP A ;累加器高低字节交换指令
MOV 24H,A ;保存数据到 RAM 的 23H 单元，数据应该为 F0H
RET
END
```

在该程序中，首先为累加器 A 和 RAM 的 20H 单元进行赋值，通过 XCH 指令来进行直接寻址的整字节交换指令。接着，为寄存器 R2 赋值，通过 XCH 指令来进行寄存器寻址的整字节交换指令。为寄存器 R0 赋值，通过 XCH 指令来进行寄存器间接寻址的整字节交换指令。为寄存器 R1 和累加器 A 赋值，通过 XCHD 指令来进行半字节数据交换指令。最后，为累加器 A 赋值，通过 SWAP 指令来进行累加器高低半字节交换指令。数据交换的结果分别保存在片内 RAM 的 20H~24H 单元。

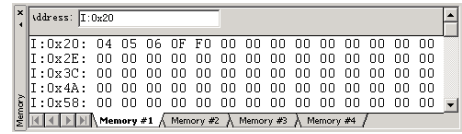


图 16-22 程序仿真结果

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以看到数据交换指令执行后，RAM 存储单元 20H~24H 中数值分别变为 0x04H、0x05H、0x06H、0x0FH 和 0xF0H，如图 16-22 所示。

16.3.5 堆栈操作指令

堆栈操作指令是对堆栈进行操作的指令。堆栈操作指令可以实现对数据或断点地址的保护，常用在子程序调用的地方进行现场保护和恢复。51 系列单片机指令系统中有以下两条堆栈操作指令。

➤ 进栈指令 PUSH

进栈指令 PUSH 是堆栈操作的入口。其执行的操作是先将栈指针 SP 的内容加 1，使其指向堆栈顶的空单元，再将直接地址 direct 单元内容送入栈顶空单元。进栈指令 PUSH 的格式如下：

```
PUSH direct ;SP←( SP )+1 , ( SP )←( direct )
```

➤ 出栈指令 POP

出栈指令 POP 是堆栈操作的出口。出栈指令执行的操作是先将 SP 所指向的单元的内容送入直接地址所指的单元，再将栈指针 SP 的内容减 1，使其指向新的栈顶空单元。出栈指令 POP 的格式如下：

```
POP direct ;direct←( ( SP ) ) , direct←( SP )-1
```

堆栈操作指令中的操作数只能以直接寻址方式取得，不能使用累加器 A 或寄存器 Rn 作为操作数。如果必须将累加器 A 的内容送入堆栈，应使用如下指令。

```
PUSH ACC
```

这里的“ACC”表示累加器 A 的直接地址 E0H。

下面通过数据的保护和恢复来讲解进栈指令 PUSH 和出栈指令 POP 在程序中的应用。程序示例如下：

```
ORG 0000H
JMP START
ORG 1000H
START: MOV 20H, #20H ;RAM 单元赋值
      PUSH 20H ;进栈
      MOV 20H, #21H ;RAM 单元数值改变
      POP 20H ;出栈
      MOV A, #30H ;累加器 A 赋值
      PUSH ACC ;进栈
      MOV A, #03H ;累加器 A 改变数值
      POP ACC ;出栈
      RET
      END
```

在该程序中，首先为 RAM 的 20H 单元赋值 0x20H，使用 PUSH 指令执行进栈操作，接着改变该单元的数值，然后通过 POP 指令执行出栈操作，此时 RAM 的 20H 单元的数值恢复为

20H。最后，还演示了累加器 A 的进栈和出栈操作。

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以看到进栈和出栈操作指令执行后，RAM 的 20H 单元及累加器 A 中的数据变化，仿真结果如图 16-23 所示。

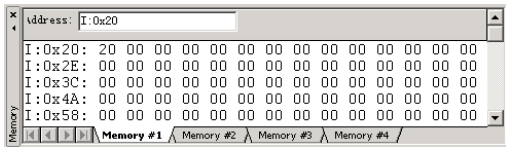


图 16-23 程序仿真结果

16.4 算术运算指令

算术运算指令是指用于数据的算术运算的操作指令。51 系列单片机的算术运算指令可以完成加、减、乘、除等算术运算操作。这些指令的操作数都是 8 位无符号数，不能直接对有符号数和 16 位的数据进行运算。

算术运算指令的执行一般会影响到程序状态字 PSW 中的一些标志位，具体表现在如下几个方面。

- 溢出标志 OV，利用 OV 可进行对带符号数进行补码运算，从而扩展指令的运算范围。
- 进位（借位）标志 CY，利用 CY 可以编写子程序，进行 16 位无符号整数的加、减运算，从而扩展指令的运算范围。
- 辅助进位标志 AC，AC 一般用于 BCD 码运算中的调整。
- 奇偶标志位 P。

51 系列单片机的指令系统中算术运算指令共有 24 条，下面按照运算的类型不同，分别进行介绍。

16.4.1 加法指令

加法指令是实现算术加法运算的操作指令。这组指令的功能是把源操作数所指的内容与累加器 A 的内容相加，其结果存放于累加器 A 中。源操作数可以采用以下 4 种寻址方式。

- 立即寻址。示例如下：

```
ADD A, #data ; A ← ( A ) + data
```

- 直接寻址。示例如下：

```
ADD A, direct ; A ← ( A ) + ( direct )
```

- 寄存器寻址。示例如下：

```
ADD A, Rn ; A ← ( A ) + ( Rn )
```

- 寄存器间接寻址。示例如下：

```
ADD A, @Ri ; A ← ( A ) + ( ( Ri ) )
```

这些指令的运算结果对程序状态字 PSW 中的运算标志位都有影响，下面举例进行说明。指令示例如下：

```
ADD A, #67H
```

假定指令操作前，累加器 A 中的数据为 4AH (01001010B)，它和立即数 67H (01100111B) 进行相加，即 4AH (0100 1010B) + 67H (0110 0111B) = B1 (1011 0001B)。

- 进位（借位）标志 CY：在加法运算中，如果 D7 位向上有进位，则 CY=1；否则 CY=0。在这个例子中，没有进位，因此 CY=0。
- 半进位标志 AC：在加法运算中，如果低 4 位向高 4 位有进位，则 AC=1；否则 AC=0。在这个例子中，存在低 4 位向高 4 位的进位，因此 AC=1。

- 溢出标志 OV：在加法运算中，如果 D7、D6 位只有一个向上有进位，则 OV=1；否则 CY=0。如果 D7、D6 位同时有进位或无进位时，则 OV=0。在这个例子中，D6 位向 D7 位有进位，而 D7 没有向外进位，因此，OV=1。
- 奇偶标志位 P：当 A 中“1”的个数为奇数时，P=1；“1”的个数为偶数时，P=0。在这个例子中，运算得到的结果为 B1（1011 0001B），其中共 4 个“1”，因此 P=0。

！注意：无符号数用 CY 表示进位、溢出（不考虑 OV 位），有符号数用 OV 表示溢出（不考虑 CY 位）。如果两个正数或两个负数相加时，发生溢出，将改变结果的符号位，所有结果都是错误的，(OV)=1 指出了这一类错误。编程人员应根据这些标志位了解当前运算结果所处的状态，以确定程序的走向。

下面分别采用上述几种加法指令来演示其在具体的程序设计中的应用。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:    MOV      A, #10H          ;累加器 A 赋值
          ADD      A, #0FH          ;立即寻址加法指令
          MOV      20H, A           ;保存结果至片内 RAM 的 20H 单元

          MOV      A, #10H          ;累加器 A 赋值
          MOV      10H, #0AH        ;片内 RAM 的 10H 单元赋值
          ADD      A, 10H           ;直接寻址加法指令
          MOV      21H, A           ;保存结果至片内 RAM 的 21H 单元

          MOV      A, #10H          ;累加器 A 赋值
          MOV      R0, #0BH         ;寄存器 R0 赋值
          ADD      A, R0            ;寄存器寻址加法指令
          MOV      22H, A           ;保存结果至片内 RAM 的 22H 单元

          MOV      A, #10H          ;累加器 A 赋值
          MOV      11H, #0CH        ;片内 RAM 的 11H 单元赋值
          MOV      R1, #11H         ;寄存器 R1 赋值
          ADD      A, @R1           ;寄存器间接寻址加法指令
          MOV      23H, A           ;保存结果至片内 RAM 的 23H 单元
          JMP      START           ;跳转
          END
```

在该程序中，首先为所使用到的累加器、RAM 单元及寄存器赋值，然后分别采用立即寻址、直接寻址、寄存器寻址和寄存器间接寻址的加法指令执行加法运算。运算完毕后，将结果保存在片内 RAM 的 20H、21H、22H 和 23H 单元。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步加法指令执行后，累加器 A 中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-24 所示。

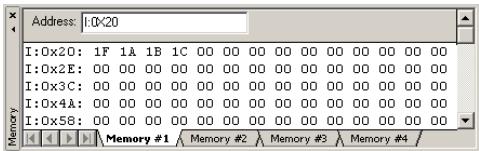


图 16-24 程序仿真结果

16.4.2 带进位的加法指令

带进位的加法指令是把源操作数、累加器 A 中的内容及进位标志相加的运算指令。其中进位标志是程序状态字 PSW 中的进位标志位 CY 的值，最终的运算结果将存放于累加器 A 中。源操作数可以采用以下 4 种寻址方式。

➤ 立即寻址。示例如下：

```
ADDC A, #data ;A← ( A ) +data+ ( CY )
```

➤ 直接寻址。示例如下：

```
ADDC A, direct ;A← ( A ) + ( direct ) + ( CY )
```

➤ 寄存器寻址。示例如下：

```
ADDC A, Rn ;A← ( A ) + ( Rn ) + ( CY )
```

➤ 寄存器间接寻址。示例如下：

```
ADDC A, @Ri ;A← ( A ) + ( ( Ri ) ) + ( CY )
```

带进位的加法指令 ADDC 的运算结果对程序状态字 PSW 各标志位的影响，与加法指令“ADD”相同。需要注意的是，这里所指的进位标志位 CY 的值是在该指令执行之前已经存在的进位标志值，而不是执行该指令过程中产生的进位标志值。

例如，假定累加器 A=B2H(10110010B)，(20H)=7EH(01111110B)，指令执行前 PSW=80H，即 (CY)=1。代码如下：

```
ADDC A, 20H
```

执行结果 (A)=30H，(CY)=1，(AC)=1，(OV)=0，(P)=0。

下面分别采用上述几种加法指令来演示其在具体的程序设计中的应用。程序示例如下：

```
ORG 0000H
JMP START
ORG 1000H
START: MOV PSW, #80H ;赋进位标志 CY 为 1
MOV A, #10H ;累加器 A 赋值
ADDC A, #0FH ;立即寻址加法指令
MOV 20H, A ;保存结果至片内 RAM 的 20H 单元

MOV PSW, #80H ;赋进位标志 CY 为 1
MOV A, #10H ;累加器 A 赋值
MOV 10H, #0AH ;片内 RAM 的 10H 单元赋值
ADDC A, 10H ;直接寻址加法指令
MOV 21H, A ;保存结果至片内 RAM 的 21H 单元

MOV PSW, #80H ;赋进位标志 CY 为 1
MOV A, #10H ;累加器 A 赋值
MOV R0, #0BH ;寄存器 R0 赋值
ADDC A, R0 ;寄存器寻址加法指令
MOV 22H, A ;保存结果至片内 RAM 的 22H 单元

MOV PSW, #80H ;赋进位标志 CY 为 1
MOV A, #10H ;累加器 A 赋值
MOV 11H, #0CH ;片内 RAM 的 11H 单元赋值
MOV R1, #11H ;寄存器 R1 赋值
```

51 单片机开发与应用技术详解

```
ADDC A, @R1                ;寄存器间接寻址加法指令
MOV      23H, A             ;保存结果至片内 RAM 的 23H 单元
JMP      START             ;跳转
END
```

在该程序中，首先赋 PSW 为 0x80，即进位标志 CY 为 1。然后为所使用到的累加器、RAM 单元以及寄存器赋值，然后分别采用立即寻址、直接寻址、寄存器寻址和寄存器间接寻址的带进位加法指令执行加法运算。运算完毕后，将结果保存在片内 RAM 的 20H、21H、22H 和 23H 单元。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，累加器 A 中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-25 所示。

另外，带进位的加法指令多用于多字节加法运算。因为低位字节相加时可能产生进位，而在进行高位字节相加时，要考虑低位字节向高位字节的进位。因此必须使用带进位的加法指令。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:    MOV  A, #56H        ;取一个加数的低字节送 A 中
          ADD  A, #0ABH       ;两个低字节数相加
          MOV  41H, A         ;结果存放 41H 单元
          MOV  A, #34H        ;取一个加数的高字节送 A 中
          ADDC A, #78H        ;高字节数相加，并加上低字节的进位
          MOV  40H, A         ;结果存放 40H 单元
          JMP  START         ;跳转
          END
```

在该程序中，有两个无符号 16 位二进制数 3456H 和 78ABH，程序将这两者相加，并将结果存于 40H、41H 单元，高字节在前，低字节在后。由于不存在 16 位数加法指令，所以只能先将低 8 位相加，后将高 8 位相加。高 8 位在相加的时候需要同时将低 8 位的进位一起相加。程序执行的示意图，如图 16-26 所示。

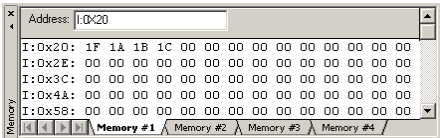


图 16-25 程序仿真结果

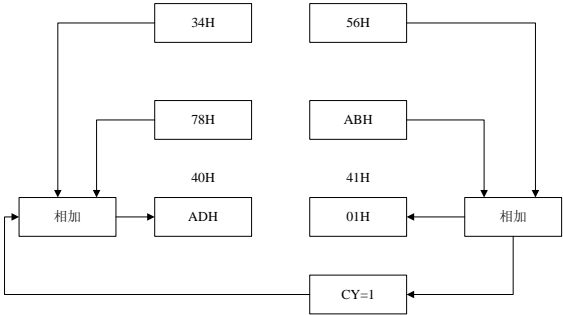


图 16-26 无符号 16 位二进制数加法示意图

16.4.3 带借位的减法指令

带借位的减法指令是将累加器 A 中的数据减去源操作数和借位标志位的运算指令。其中，借位标志位为 CY，其运算结果存放于累加器 A 中。源操作数可以采用以下 4 种寻址方式。

➤ 立即寻址。示例如下：

```
SUBB A, #data                ;A← ( A ) -data- ( CY )
```

➤ 直接寻址。示例如下：

```
SUBB A,direct          ;A←(A)-(direct)-(CY)
```

➤ 寄存器寻址。示例如下：

```
SUBB A,Rn              ;A←(A)-(Rn)-(CY)
```

➤ 寄存器间接寻址。示例如下：

```
SUBB A,@Ri             ;A←(A)-((Ri))-(CY)
```

该指令运算后，会影响程序状态字 PSW 中的标志位。下面举例进行说明。

```
SUBB A,#74H
```

在该指令执行前，累加器 A 中的数据为 DCH (11011100B)，CY=1。指令执行的操作为 DCH (11011100B) -74H (01110100B) -1=67H (01100111B)。

- 借位标志 CY：在减法运算中，如果 D7 位向上有借位，则 CY=1；否则 CY=0。在本例中，没有产生借位，因此 CY=0。
- 半借位标志 AC：在减法运算中，如果 D3 位向上有借位，则 AC=1；否则 AC=0。在本例中，D3 位没有向上借位，因此 AC=0。
- 溢出标志 OV：在减法运算中，如果 D7、D6 位只有一个向上有借位，则 OV=1；否则 CY=0；，如果 D7、D6 位同时有借位或无借位时，则 OV=0。在本例中，D6 向 D7 位借位，而 D7 没有借位，因此 OV=1。
- 奇偶标志位 P：当 A 中“1”的个数为奇数时，P=1；“1”的个数为偶数时，P=0。在本例中，P=1。
- 在使用带借位的减法指令时，需要注意以下几点。
- 51 系列单片机指令系统中只有带借位减法指令，没有不带借位的减法指令。若要进行不带借位的减法运算，应先将 CY 清零，然后再执行本指令。
- 在单片机 CPU 内部进行减法运算时，实际上是将操作数变成补码，然后再相加。
- 如果两个带符号数相减时，有时会发生溢出，将改变结果的符号位，这样的结果是错误的，(OV)=1 便指出了这一类错误。

下面分别采用上述几种带借位的减法指令来演示其在具体的程序设计中的应用。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      PSW,#80H          ;赋借位标志 CY 为 1
          MOV      A,#0DCH          ;累加器 A 赋值
          SUBB A,#74H               ;立即寻址减法指令
          MOV      20H,A            ;保存结果至片内 RAM 的 20H 单元

          MOV      PSW,#80H          ;赋借位标志 CY 为 1
          MOV      A,#0DCH          ;累加器 A 赋值
          MOV      10H,#74H         ;片内 RAM 的 10H 单元赋值
          SUBB A,10H                ;直接寻址减法指令
          MOV      21H,A            ;保存结果至片内 RAM 的 21H 单元

          MOV      PSW,#80H          ;赋借位标志 CY 为 1
          MOV      A,#0DCH          ;累加器 A 赋值
          MOV      R0,#74H          ;寄存器 R0 赋值
          SUBB A,R0                 ;寄存器寻址减法指令
          MOV      22H,A            ;保存结果至片内 RAM 的 22H 单元
```

```
MOV      PSW, #80H      ; 赋值标志 CY 为 1
MOV      A, #0DCH      ; 累加器 A 赋值
MOV      11H, #74H      ; 片内 RAM 的 11H 单元赋值
MOV      R1, #11H      ; 寄存器 R1 赋值
SUBB A, @R1             ; 寄存器间接寻址减法指令
MOV      23H, A         ; 保存结果至片内 RAM 的 23H 单元
JMP      START          ; 跳转
END
```

在该程序中，首先赋 PSW 为 0x80，即借位标志 CY 为 1。然后为所使用到的累加器、RAM 单元及寄存器赋值，然后分别采用立即寻址、直接寻址、寄存器寻址和寄存器间接寻址的带借位减法指令执行加法运算。运算完毕后，将结果保存在片内 RAM 的 20H、21H、22H 和 23H 单元。

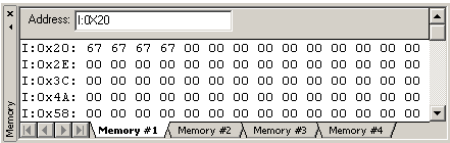


图 16-27 程序仿真结果

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，累加器 A 中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-27 所示。

16.4.4 加 1 指令

加 1 指令是把源操作数所指向单元中的数据加 1 的运算指令。其中源操作数可以有以下三种寻址方式。

➤ 直接寻址。示例如下：

```
INC direct      ; direct ← ( direct ) + 1
```

➤ 寄存器寻址。示例如下：

```
INC Rn          ; Rn ← ( Rn ) + 1
INC DPTR        ; DPTR ← ( DPTR ) + 1
```

➤ 寄存器间接寻址。这种方式常用于修改操作数的地址，示例如下：

```
INC @Ri         ; Ri ← ( Ri ) + 1
```

以上这些指令都不会影响程序状态字 PSW 中的运算标志位，如果涉及累加器 A，示例如下：

```
INC A           ; A ← ( A ) + 1
```

这个指令会影响 PSW 中的 P 标志位。

例如，单片机 RAM 的 20H 单元中的数据为 67H，执行以下指令后，该单元中的数据变为 68H。

```
INC 20H
```

又如，累加器 A 中的数据为 34H (00110100B)，P=1；执行以下指令后，该单元中的数据变为 35H (00110101B)，并且 P=0。

```
INC A
```

下面分别采用上述几种寻址方式的加 1 指令来演示其在具体的程序设计中的应用。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:   MOV      20H, #74H      ; 片内 RAM 的 20H 单元赋值
INC      20H                   ; 直接寻址加 1 指令
```



```
MOV      R0, #23H          ;寄存器 R0 赋值
INC      R0                ;寄存器寻址加 1 指令
MOV      21H, R0           ;保存结果

MOV      A, #12H           ;累加器 A 赋值
INC      A                 ;寄存器寻址加 1 指令
MOV      22H, A            ;保存结果

MOV      10H, #67H         ;RAM 单元赋值
MOV      R1, #10H         ;寄存器赋值
INC      @R1               ;寄存器间接寻址加 1 指令
MOV      23H, @R1         ;保存结果
JMP      START             ;跳转
END
```

在该程序中，首先为所使用到的累加器、RAM 单元及寄存器赋值，然后分别采用直接寻址、寄存器寻址和寄存器间接寻址的加 1 指令执行运算。运算完毕后，将结果保存在片内 RAM 的 20H、21H、22H 和 23H 单元。

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在每一步加 1 指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-28 所示。

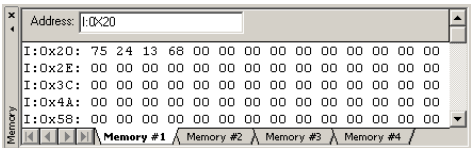


图 16-28 程序仿真结果

16.4.5 减 1 指令

减 1 指令是把源操作数所指的单元的内容减 1 的运算指令。其中源操作数可以有以下三种寻址方式。

➤ 直接寻址。示例如下：

```
DEC direct          ;direct←(direct) +1
```

➤ 寄存器寻址。示例如下：

```
DEC Rn              ;Rn←(Rn) +1
DEC DPTR            ;DPTR←(DPTR) +1
```

➤ 寄存器间接寻址。这种方式常用于修改操作数的地址，示例如下：

```
DEC @Ri             ;Ri←(Ri) +1
```

同加 1 指令一样，这组指令都不影响 PSW 的标志位，而对于累加器的操作指令“DEC A”则影响 P 标志位。下面分别采用上述几种寻址方式的减 1 指令来演示其在具体的程序设计中的应用。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:   MOV      20H, #74H          ;片内 RAM 的 20H 单元赋值
DEC      20H                       ;直接寻址减 1 指令

MOV      R0, #23H                  ;寄存器 R0 赋值
DEC      R0                        ;寄存器寻址减 1 指令
MOV      21H, R0                   ;保存结果
```

```
MOV      A, #12H           ;累加器 A 赋值
DEC      A                 ;寄存器寻址减 1 指令
MOV      22H, A            ;保存结果

MOV      10H, #67H         ;RAM 单元赋值
MOV      R1, #10H          ;寄存器赋值
DEC      @R1               ;寄存器间接寻址减 1 指令
MOV      23H, @R1          ;保存结果
JMP      START             ;跳转
END
```

在该程序中，首先为所使用到的累加器、RAM 单元及寄存器赋值，然后分别采用直接寻址、寄存器寻址和寄存器间接寻址的减 1 指令执行运算。运算完毕后，将结果保存在片内 RAM 的 20H、21H、22H 和 23H 单元。

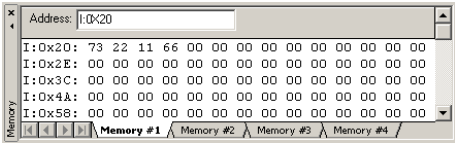


图 16-29 程序仿真结果

将程序在 Keil μ Vision3 中进行编译,通过单步仿真调试,可以查看程序在每一步减 1 指令执行后,寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-29 所示。

16.4.6 乘除法指令

乘除法指令是执行算术乘和除的运算指令。51 系列单片机的指令系统中,乘除法指令各有一个,用于完成 8 位源操作数的乘除法运算。这两个指令都是 1 字节指令,执行时需 4 个机器周期,很占用 CPU 时间。

1. 乘法指令

乘法指令是把累加器 A 和寄存器 B 中的两个数据相乘的运算指令。其中, A 和 B 中的数据均为 8 位无符号数,所得的乘积结果为 16 位无符号数据。结果的低 8 位存放在累加器 A 中,高 8 位存放在寄存器 B 中。指令的格式如下:

```
MUL AB                      ;BA←(A)×(B)
```

乘法指令的执行会影响以下三个标志位。

- 进位(借位)标志 CY 总是被清零;
- 溢出标志 OV: 若乘积小于 FFH(即 B 的内容为 0),则 OV=0, 否则 OV=1;
- 奇偶标志 P 仍按计算结果 A 中“1”的个数来确定,为奇数则 P=1, 否则 P=0。

例如,假设 (A)=91H, (B)=3AH。

```
MUL AB
```

指令执行后 (A)=DAH, (B)=20H, OV=1, CY=0, P=1。

2. 除法指令

除法指令是把累加器 A 和寄存器 B 中的两个数据相除的运算指令。除法指令在执行前,需要将被除数放在累加器 A 中,将除数放在寄存器 B 中,被除数和除数都是 8 位无符号数。指令执行完毕后,将所得的结果的商放在累加器 A 中,余数放在寄存器 B 中。指令的格式如下:

```
DIV AB                      ;A←商, B←余数
```

除法指令的执行会影响以下三个标志。

- 进位(借位)标志 CY 总是被清零;
- 溢出标志 OV: 若除数为 0(即 B 的内容为 0),则 OV=1, 否则 OV=0; OV=1 表示该除法没有意义, OV=0 表示除法有意义;
- 奇偶标志 P 仍按计算结果 A 中“1”的个数来确定,为奇数则 P=1, 否则 P=0。

例如，假设 (A) =8AH (138D)，(B) =1BH (27D)。

DIV AB

指令执行后 (A) =05H，(B) =03H，OV=0，CY=0，P=0。

下面分别采用乘法指令和除法指令来讲解其在具体程序中的应用，程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      A, #12H      ;累加器 A 赋值
          MOV      B, #05H      ;寄存器 B 赋值
          MUL      AB          ;乘法运算
          MOV      20H, B        ;保存高 8 位结果
          MOV      21H, A        ;保存低 8 位结果

          MOV      A, #12H      ;累加器 A 赋值
          MOV      B, #05H      ;寄存器 B 赋值
          DIV      AB          ;除法运算
          MOV      22H, A        ;保存商
          MOV      23H, B        ;保存余数
          JMP      START        ;跳转
END
```

在该程序中，首先为累加器 A 和寄存器 B 赋值，然后使用 MUL 指令执行乘法运算，并将运算的结果高 8 位保存在片内 RAM 的 20H 单元，低 8 位结果保存在 21H 单元。接着，同样为累加器 A 和寄存器 B 赋值，通过 DIV 指令执行除法运算，并将运算的结果的商和余数分别保存到 22H 单元和 23H 单元。

将程序在 Keil μVision3 中进行编译,通过单步仿真调试,可以查看程序在每一步指令执行后,寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-30 所示。

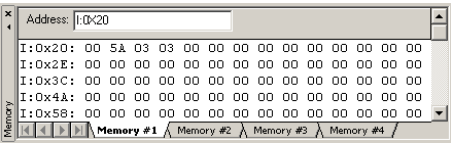


图 16-30 程序仿真结果

16.4.7 十进制调整指令

十进制调整指令是对累加器 A 中由前两个变量（每一个变量均为压缩的 BCD 码）的加法结果进行 BCD 码修正的指令。指令执行的结果将变为 BCD 码的数。指令的格式如下：

DA A

51 系列单片机的指令系统中的加法运算及存储是按照二进制规则进行的,如果希望单片机进行十进制数的运算,并且储存十进制数,此时就需要用 BCD 码来表示十进制数。

BCD 码就是采用 4 位二进制编码来表示十进制数。4 位二进制数共有 16 个编码,即 0000~1111,BCD 码是取其前 10 个编码(即 0000~1001)来表示的十进制数 0~9,这种编码称为“8421BCD 码”,简称为“BCD 码”。这样一个字节可以存放两位 BCD 码,这是一种压缩的 BCD 码。

对于“BCD 码加法”,两个 BCD 码数据相加,结果也应该是 BCD 码。单片机的加法指令在运算时,对于 4 位二进制数是逢 16 进位;而 BCD 码是逢 10 进位,这种进位差将导致当运算结果大于 16 或在 10 与 16 之间时,会得到错误结果,即得到的不是 BCD 码数。因此要对结果进行修正,即进行十进制调整。

使用十进制调整指令可以自动对运算结果进行调整。单片机在执行十进制调整指令时,由 ALU 硬件中的十进制调整电路自动进行的,其执行的操作示意如图 16-31 所示。只需要在加法指令后加上十进制调整指令即可。

51 单片机开发与应用技术详解

由执行过程中可以看出，该指令只影响进位标志 CY。在执行“DAA”指令之后，若 CY=1，则表明 BCD 码加法结果已经大于等于十进制数 100。

下面通过举例来讲解十进制调整指令在具体程序中的应用。这里假定需要进行 90+64 的 BCD 码加法，并将结果存入 20H、21H 单元，高位在后，低位在前。程序示例如下：

```

      ORG      0000H
      JMP      START
      ORG      1000H
START:  MOV     A, #90H           ;90 的 BCD 码送 A
      ADD     A, #64H           ;A 与 64 的 BCD 码相加的结果送 A
      DA      A                 ;对结果进行十进制调整
      MOV     20H, A            ;A 中的结果存入 20H
      MOV     A, #00H          ;A 清零
      ADDC    A, #00H           ;加进位
      DA      A                 ;对结果进行十进制调整
      MOV     21H, A            ;存进位
      JMP     START             ;跳转
      END
```

在该程序中，首先使用 ADD 指令执行加法运算，然后通过 DA 指令执行十进制调整。指令执行的结果是 A=54H，CY=1。最后分别将结果保存在片内 RAM 的 20H 和 21H 单元。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-32 所示。

另外，需要说明的是，十进制调整指令“DA A”只能跟在加法指令后面，51 系列单片机的指令系统中没有十进制减法调整指令。要进行十进制减法运算，可以取减数的补数来进行相加，两位十进制数是对 100 取补的。例如，50-20=30，改为补数相加为 50+（100-20）=130，舍去进位 1 后，就得到正确的结果。

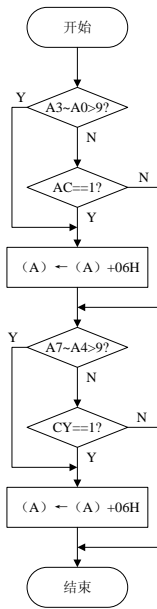


图 16-31 十进制调整指令执行示意图

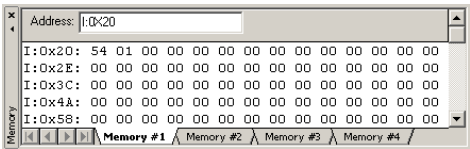


图 16-32 程序仿真结果

16.5 逻辑运算及移位指令

逻辑运算指令是进行逻辑运算的操作指令。51 系列单片机中的逻辑运算包括与、或和异或三类指令，每类各有 6 条指令。另外，还有累加器 A 清零指令一条、取反指令一条，以及移位指令 4 条。逻辑运算及移位指令共有 24 条指令，这些指令都是按位进行的。下面分别介绍这些指令。

一般来说，逻辑运算指令对程序状态字 PSW 中的运算标志位 CY、AC 和 OV 都没有影响，只有涉及累加器 A 时，才会影响奇偶标志位 P。

16.5.1 逻辑与指令

逻辑与指令是将两个源操作数按二进制展开，并按对应的位进行与运算的指令。下面用符号“ \wedge ”来表示逻辑与。逻辑与运算的指令按照运算结果存放的位置可以分为以下两类。

1. 存放在累加器 A 中

```
ANL    A,    #data           ; $A \leftarrow (A) \wedge data$ 
ANL    A,    direct         ; $A \leftarrow (A) \wedge (direct)$ 
ANL    A,    Rn             ; $A \leftarrow (A) \wedge (Rn)$ 
ANL    A,    @Ri            ; $A \leftarrow (A) \wedge ((Ri))$ 
```

这 4 条指令执行时，将累加器 A 的内容和源操作数所指的内容按位相与，逻辑与的结果存放在累加器 A 中。

2. 存放在直接寻址单元中

```
ANL    direct, #data         ; $direct \leftarrow (direct) \wedge data$ 
ANL    direct, A             ; $direct \leftarrow (direct) \wedge (A)$ 
```

这两条指令执行时，是将直接地址单元中的内容和源操作数所指的内容按位相与，逻辑与的结果存放在直接寻址所指定的单元中。

下面分别采用上述 6 个指令来讲解逻辑与指令在具体程序中的应用，程序示例如下：

```
ORG    0000H
JMP    START
ORG    1000H
START:  MOV    A, #5AH        ;寄存器 A 赋值
        ANL    A, #23H        ;立即寻址逻辑与运算
        MOV    20H, A         ;保存运算结果

        MOV    10H, #07H      ;片内 RAM 单元赋值
        MOV    A, #5AH        ;寄存器 A 赋值
        ANL    A, 10H         ;直接寻址逻辑与运算
        MOV    21H, A         ;保存运算结果

        MOV    R0, #30H       ;寄存器赋值
        MOV    A, #5AH        ;寄存器 A 赋值
        ANL    A, R0          ;寄存器寻址逻辑与运算
        MOV    22H, A         ;保存运算结果

        MOV    11H, #22H      ;寄存器赋值
        MOV    R1, #11H       ;寄存器赋值
        MOV    A, #5AH        ;寄存器 A 赋值
        ANL    A, @R1         ;寄存器间接寻址逻辑与运算
```

```
MOV      23H,A           ;保存运算结果

MOV      24H,#89H        ;片内 RAM 赋值
ANL      24H,#34H        ;逻辑与运算

MOV      25H,#89H        ;片内 RAM 赋值
MOV      A,#5AH          ;寄存器 A 赋值
ANL      25H,A           ;逻辑与运算
JMP      START          ;跳转
END
```

在该程序中，首先为所使用到的寄存器及 RAM 单元赋值，然后分别采用上述 6 种逻辑与指令进行与运算，运算的结果保存在片内 RAM 的 20H~25H 单元。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-33 所示。

逻辑与运算指令也常用于将某些位屏蔽，只要将需要屏蔽的位和“0”相与，要保留的位和“1”相与即可。另外，如果使用此指令修改一个输出口时，作为原始数据的值将从单片机的输出数据口锁存器（P0~P3）读入，而不是读引脚的状态。

16.5.2 逻辑或指令

逻辑或运算是将两个源操作数按二进制展开，并按对应的位进行或运算的操作指令。下面用符号“∨”来表示逻辑或。逻辑或运算的指令按照运算结果存放的位置可以分为以下两类。

1. 存放在累加器 A 中

```
ORL      A, #data        ;A←(A)∨data
ORL      A, direct       ;A←(A)∨(direct)
ORL      A, Rn           ;A←(A)∨(Rn)
ORL      A, @Ri          ;A←(A)∨(Ri)
```

这 4 条指令执行时，将累加器 A 的内容和源操作数所指的内容按位相或，逻辑或的结果存放在累加器 A 中。

2. 存放在直接寻址单元中

```
ORL      direct, #data    ;direct←(direct)∨data
ORL      direct, A        ;direct←(direct)∨(A)
```

这两条指令执行时，是将直接地址单元中的内容和源操作数所指的内容按位相或，逻辑或的结果存放在直接寻址所指定的单元中。

逻辑或运算指令常用于将某些位置位，即使之为“1”，只要将需要置位的位和“1”相或，要保留的位和“0”相或即可。例如，要将累加器 A 的高 4 位送到 P2 端口的高 4 位输出，而 P2 端口的低 4 位保持不变。程序示例如下：

```
ANL      A, #0F0H        ;屏蔽低 4 位，高 4 位不变
ANL      P2, #0FH        ;屏蔽 P1 高 4 位，低 4 位不变
ORL      P2, A           ;执行或运算
```

另外，如果使用此指令修改一个输出口时，作为原始数据的值将从单片机的输出数据口锁存器（P0~P3）读入，而不是读引脚的状态。下面分别采用上述 6 个指令来讲解逻辑或指令在具体程序中的应用，程序示例如下：

```
ORG      0000H
JMP      START
```

```

ORG      1000H
START:   MOV      A, #5AH      ; 寄存器 A 赋值
        ORL       A, #23H     ; 立即寻址逻辑或运算
        MOV       20H, A      ; 保存运算结果

        MOV       10H, #07H   ; 片内 RAM 单元赋值
        MOV       A, #5AH     ; 寄存器 A 赋值
        ORL       A, 10H      ; 直接寻址逻辑或运算
        MOV       21H, A      ; 保存运算结果

        MOV       R0, #30H    ; 寄存器赋值
        MOV       A, #5AH     ; 寄存器 A 赋值
        ORL       A, R0       ; 寄存器寻址逻辑或运算
        MOV       22H, A      ; 保存运算结果

        MOV       11H, #22H   ; 寄存器赋值
        MOV       R1, #11H    ; 寄存器 A 赋值
        MOV       A, #5AH     ; 寄存器 A 赋值
        ORL       A, @R1      ; 寄存器间接寻址逻辑或运算
        MOV       23H, A      ; 保存运算结果

        MOV       24H, #89H   ; 片内 RAM 赋值
        ORL       24H, #34H   ; 逻辑或运算

        MOV       25H, #89H   ; 片内 RAM 赋值
        MOV       A, #5AH     ; 寄存器 A 赋值
        ORL       25H, A      ; 逻辑或运算
        JMP       START       ; 跳转
END
```

在该程序中，首先为所使用到的寄存器及 RAM 单元赋值，然后分别采用上述 6 种逻辑或指令进行或运算。运算的结果保存在片内 RAM 的 20H~25H 单元。

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-34 所示。

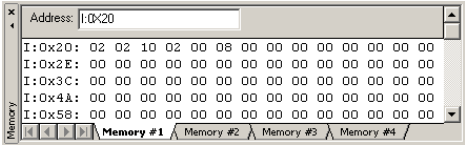


图 16-33 程序仿真结果

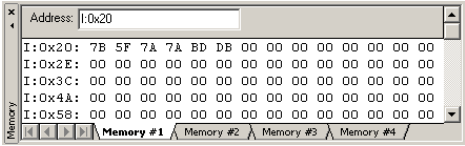


图 16-34 程序仿真结果

16.5.3 逻辑异或指令

逻辑异或运算是将两个源操作数按二进制展开，并按对应的位进行异或运算的操作指令，下面用符号“ \oplus ”来表示逻辑异或。逻辑异或运算的指令按照运算结果存放的位置可以分为以下两类。

1. 存放在累加器 A 中

```
XRL      A, #data      ; A ← (A)  $\oplus$  data
```

```
XRL      A,   direct           ; A← ( A ) ⊕ ( direct )
XRL      A,   Rn              ; A← ( A ) ⊕ ( Rn )
XRL      A,   @Ri             ; A← ( A ) ⊕ ( ( Ri ) )
```

这 4 条指令执行时，将累加器 A 的内容和源操作数所指的内容按位相异或，逻辑异或的结果存放在累加器 A 中。

2. 存放在直接寻址单元中

```
XRL      direct, #data         ; direct← ( direct ) ⊕ data
XRL      direct, A             ; direct← ( direct ) ⊕ ( A )
```

这两条指令执行时，是将直接地址单元中的内容和源操作数所指的内容按位相异或，逻辑异或的结果存放在直接寻址所指定的单元中。

逻辑异或运算指令常用于将某些位取反，即使“0”的位变为“1”，使“1”的位变为“0”。只要将需要求反的位和“1”异或，要保留的位和“0”异或即可。

例如，将内部累加器 A 中的 D2~D5 位取反，只需执行下面的指令即可。

```
XRL      A,      #3CH
```

另外，如果使用此指令修改一个输出口时，作为原始数据的值将从单片机的输出数据口锁存器（P0~P3）读入，而不是读引脚的状态。下面分别采用上述 6 条指令来讲解逻辑异或指令在具体程序中的应用，程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      A, #5AH           ; 寄存器 A 赋值
XRL      A, #23H                   ; 立即寻址逻辑异或运算
MOV      20H, A                    ; 保存运算结果

MOV      10H, #07H                 ; 片内 RAM 单元赋值
MOV      A, #5AH                   ; 寄存器 A 赋值
XRL      A, 10H                    ; 直接寻址逻辑异或运算
MOV      21H, A                    ; 保存运算结果

MOV      R0, #30H                  ; 寄存器赋值
MOV      A, #5AH                   ; 寄存器 A 赋值
XRL      A, R0                     ; 寄存器寻址逻辑异或运算
MOV      22H, A                    ; 保存运算结果

MOV      11H, #22H
MOV      R1, #11H                  ; 寄存器赋值
MOV      A, #5AH                   ; 寄存器 A 赋值
XRL      A, @R1                    ; 寄存器间接寻址逻辑异或运算
MOV      23H, A                    ; 保存运算结果

MOV      24H, #89H                 ; 片内 RAM 赋值
XRL      24H, #34H                 ; 逻辑异或运算

MOV      25H, #89H                 ; 片内 RAM 赋值
MOV      A, #5AH                   ; 寄存器 A 赋值
```



```
XRL      25H,A      ;逻辑异或运算
JMP      START      ;跳转
END
```

在该程序中，首先为所使用到的寄存器及 RAM 单元赋值，然后分别采用上述 6 种逻辑异或指令进行异或运算。运算的结果保存在片内 RAM 的 20H~25H 单元。

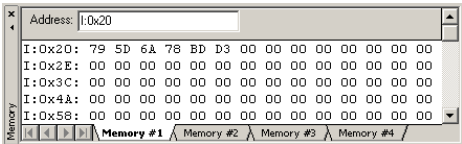


图 16-35 程序仿真结果

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-35 所示。

16.5.4 累加器清零指令

累加器清零指令是将累加器 A 中的数据进行清零的操作指令。累加器清零指令只有一条，为单字节指令，其格式如下：

```
CLR      A          ;A←0
```

该指令只是简单地将累加器 A 的每一个二进制位置零，这将影响标志位 P。下面通过实例来讲解累加器清零指令在具体程序中的应用，程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      A,#5AH      ;寄存器 A 赋值
          MOV      20H,A      ;保存结果
          CLR      A          ;累加器清零
          MOV      21H,A      ;保存结果
          JMP      START      ;跳转
END
```

在该程序中，首先为累加器 A 赋值，然后将结果保存至片内 RAM 的 20H 单元；接着使用 CLR 指令将累加器清零，并将结果保存至 21H 单元。

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在 CLR 指令执行后，累加器 A 中的内容已经清零，因此 RAM 的 21H 单元的数据为零。程序仿真结果如图 16-36 所示。

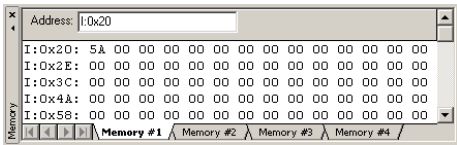


图 16-36 程序仿真结果

16.5.5 累加器取反指令

累加器取反指令是将累加器 A 中的数据进行取反的操作指令。累加器取反指令只有一条，为单字节指令，其格式如下：

```
CPL      A          ;A←(  $\overline{A}$  )
```

同累加器清零指令一样，这条指令将会影响标志位 P。

51 系列单片机的指令系统中只有对累加器 A 的取反指令，而没有求补指令。对于求补操作，可以采用“求反加 1”来完成，示例如下：

```
CPL      A          ;取反
INC      A          ;加 1，最终完成求补运算
```

下面通过实例来讲解累加器取反指令在具体程序中的应用，程序示例如下：

```
ORG      0000H
```

```
JMP      START
ORG      1000H

START:    MOV      A, #5AH      ;寄存器 A 赋值
          MOV      20H, A      ;保存结果
          CPL      A          ;累加器取反
          MOV      21H, A      ;保存结果
          JMP      START      ;跳转
          END
```

在该程序中，首先为累加器 A 赋值 0x5A，然后将结果保存至片内 RAM 的 20H 单元；接着使用 CPL 指令将累加器取反，并将结果保存至 21H 单元。

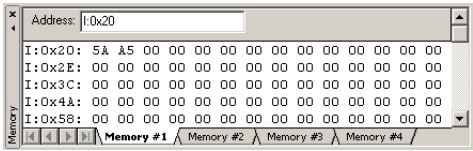


图 16-37 程序仿真结果

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在 CPL 指令执行后，累加器 A 中的内容已经取反，因此 RAM 的 21H 单元的数据为 0xA5。程序仿真结果如图 16-37 所示。

16.5.6 组合逻辑电路的实现

组合逻辑电路是由一些基本逻辑单元组合而成的，用于完成复杂逻辑运算的电路。利用以上逻辑运算指令，可以在程序中模拟各种硬件逻辑电路。

例如，对如图 16-38 所示的组合逻辑电路，编写程序模拟其功能。设输入信号 X、Y、Z 放在 20H、21H、22H 单元中，输出信号 F 放在 30H 单元。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:    MOV      20H, #61H
          MOV      21H, #0A1H
          MOV      22H, #70H

          MOV      A, 20H      ;A←( X )
          ANL      A, 21H      ;A←( A ) ∧ ( Y )
          MOV      R1, A      ;A 内容暂存
          MOV      A, 21H      ;A←( Y )
          XRL      A, 22H      ;A←( Y ) ( Z )
          CPL      A          ;A←
          ORL      A, R1      ;得到输出
          MOV      30H, A      ;保存输出结果到 30H 单元
          JMP      START      ;跳转
          END
```

在该程序中，首先为 20H、21H 和 22H 单元赋值，然后分别采用本节介绍的基本逻辑指令来完成组合逻辑电路。将程序在 Keil μ Vision3 中进行编译，通过仿真调试，可以看到逻辑电路运算的结果，如图 16-39 所示。

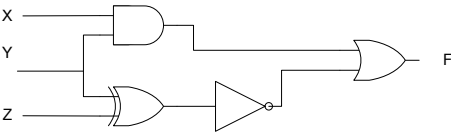


图 16-38 组合逻辑原理

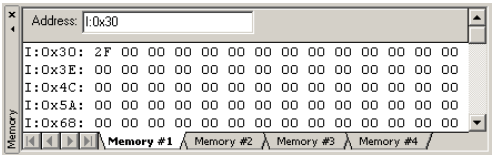


图 16-39 程序仿真结果

16.5.7 循环移位指令

循环移位指令是对累加器 A 中的内容进行循环移位的操作指令。这里要注意，51 系列单片机的移位指令只能对累加器 A 进行移位，共有以下 4 条指令。

1. 循环左移

循环左移指令是将累加器 A 的内容循环左移一位的操作指令。循环左移指令的执行过程，如图 16-40 所示。指令的格式如下。

```
RL A ; An+1←An , A0←A7
```

循环左移指令不影响 PSW 标志位。

2. 循环右移

循环右移指令是将累加器 A 的内容循环右移一位的操作指令。循环右移指令的执行过程，如图 16-41 所示。指令的格式如下。

```
RR A ; An←An+1 , A7←A0
```

循环右移指令不影响 PSW 标志位。

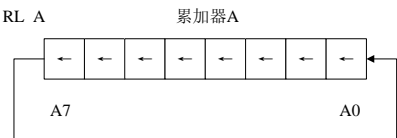


图 16-40 循环左移指令

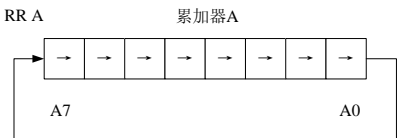


图 16-41 循环右移指令

3. 带进位循环左移

带进位循环左移指令是将累加器 A 的内容连带进位位 CY 一起循环左移 1 位的操作指令。带进位循环左移指令执行过程，如图 16-42 所示。

```
RLC A ; An+1←An , CY←A7 , A0←CY
```

带进位循环左移指令执行后影响进位位 CY 和奇偶标志位 P。

4. 带进位循环右移

带进位循环右移指令是将累加器 A 的内容连带进位位 CY 一起循环右移 1 位的操作指令。带进位循环右移指令的执行过程，如图 16-43 所示。

```
RRC A ; An←An+1 , A7←CY , CY←A0
```

带进位循环右移指令执行后影响进位位 CY 和奇偶标志位 P。

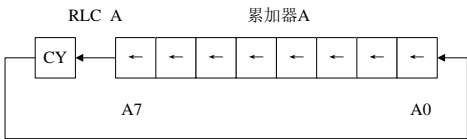


图 16-42 带进位循环左移指令

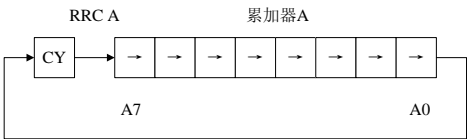


图 16-43 带进位循环右移指令

利用移位运算指令，有时可以完成乘除法的运算，比直接用乘除法指令要执行的快。例如，假设 (A)=01H。

```
RL A ; A 的内容左移 1 位，(A)=02H
RL A ; A 的内容左移 1 位，(A)=04H
RL A ; A 的内容左移 1 位，(A)=06H
.....
```

```
RL    A                                ;A 的内容左移 1 位，( A )=80H
```

由结果可见，如果源操作数小于 80H，利用左移指令每执行一次，相当于源操作数乘 2。再例如，假定 (A)=80H。

```
RR    A                                ;A 的内容右移 1 位，( A )=40H
RR    A                                ;A 的内容右移 1 位，( A )=20H
RR    A                                ;A 的内容右移 1 位，( A )=10H
.....
RR    A                                ;A 的内容右移 1 位，( A )=01H
```

由结果可见，如果源操作数小于 80H 且为偶数，则利用右移指令每执行一位，相当于源操作数除 2。

下面分别采用上述 4 种循环移位指令来讲解其在具体程序设计中的应用，程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:    MOV     A, #5AH                ;寄存器 A 赋值
          RL      A                      ;循环左移
          MOV     20H, A                  ;保存结果

          MOV     A, #5AH                ;寄存器 A 赋值
          RR      A                      ;循环右移
          MOV     21H, A                  ;保存结果

          MOV     PSW, #80H              ;CY 值 1
          MOV     A, #5AH                ;寄存器 A 赋值
          RLC     A                      ;带进位循环左移
          MOV     22H, A                  ;保存结果

          MOV     PSW, #80H              ;CY 值 1
          MOV     A, #5AH                ;寄存器 A 赋值
          RRC     A                      ;带进位循环右移
          MOV     23H, A                  ;保存结果
          JMP     START                  ;跳转
          END
```

在该程序中，首先为累加器 A 赋值，然后分别使用 RL、RR、RLC 和 RRC 指令来进行移位操作。其中对于带进位的循环移位，则首先将进位位 CY 赋为 1。最后将结果分别保存至单片机片内 RAM 的 20H、21H、22H 和 23H 单元。

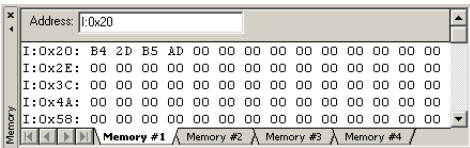


图 16-44 程序仿真结果

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-44 所示。

16.6 控制转移指令

控制转移指令是用来改变程序执行顺序的操作指令。通过它可以按照设计任务的需要来实现程序的控制转移。在 51 系列单片机的程序中，通常是按顺序进行的，程序的顺序执行是由程序计

数器 PC 自动加 1 实现的。为了适应复杂的测控应用系统的需要，经常需要改变程序的执行顺序，实现分支转向，这种情况称为“程序转移”。控制转移类指令一般是通过改变 PC 值来实现的。

本节主要讲解 51 系列单片机指令系统中的控制转移指令，包括无条件转移、有条件转移、子程序调用及返回指令。控制转移指令一般不影响标志位。

16.6.1 无条件转移指令

无条件转移指令是指不进行条件判断而强制执行的程序转移指令。通过它可以在程序中无条件转移到任何其他位置来接着执行。在 51 系列单片机中有以下 4 条无条件转移指令，分别用于不同的控制方式及程序转移长度。

1. 长转移指令

长转移指令是使用 16 位地址的程序转移指令。它使用 LJMP 指令来执行，是一个 3 字节指令。指令的格式如下：

```
LJMP addr16 ;PC ←addr16
```

长转移指令的三个字节依次是操作码（02H）、高 8 位地址、低 8 位地址。该指令的功能是把 16 位目的地址（addr16）送入程序计数器 PC，从而使程序无条件转移到 addr16 处执行。由于 16 位的数据可寻址 64KB，其程序转移范围大，可达 64KB，故称为“长转移指令”。下面采用 LJMP 指令来讲解长转移指令在具体程序中的应用，程序示例如下：

```
ORG 0000H
JMP START
ORG 1000H
START: MOV A, #5AH ;寄存器 A 赋值
      RL A ;循环左移
      MOV 20H, A ;保存结果
      MOV A, #5AH ;寄存器 A 赋值
      RR A ;循环右移
      MOV 21H, A ;保存结果
      MOV PSW, #80H ;CY 值 1
      MOV A, #5AH ;寄存器 A 赋值
      RLC A ;带进位循环左移
      MOV 22H, A ;保存结果
      MOV PSW, #80H ;CY 值 1
      MOV A, #5AH ;寄存器 A 赋值
      RRC A ;带进位循环右移
      MOV 23H, A ;保存结果
      ;
      ;其他代码段
      ;
      LJMP START ;长转移指令
      END
```

在该程序中，使用标号 START 指明跳转地址。程序代码段比较多，因此，最后采用 LJMP 指令执行长转移至 START 地址处，来实现程序的循环执行。

2. 绝对转移指令

绝对转移指令是采用 11 位跳转地址的操作指令。这是一条 2 字节指令，其指令格式如下：

```
AJMP addr11 ;PC←( PC ) +2 , PC10~0←addr11
```

该指令的功能是构造程序转移的目的地址，实现程序转移。其构造方法是：以指令提供的 11 位地址去替换 PC 的低 11 位内容，形成新的 PC 值，即转移的目的地址。

该指令在执行时，先将 PC+2，即指向下一条指令的位置；然后通过 PC 的高 5 位和指令第一个字节的高 3 位及第二个字节相连而得到的跳转目的地址送入 PC 中，从而实现跳转。指令示例如下：

```
FUN:    AJMP 0540H
```

此处 0540H=00000101 01000000B，假设标号为 FUN 处的地址为 1000H，程序计数器 PC 加 2 后的内容为 1002H，以 11 位绝对转移地址替换 PC 的低 11 位内容，则指令执行后，程序将转向 1540H 处执行。下面采用 AJMP 指令来讲解长转移指令在具体程序中的应用，程序示例如下：

```
ORG     0000H
JMP     START
ORG     0100H
START:  MOV     A, #5AH           ;寄存器 A 赋值
        RL      A               ;循环左移
        MOV     20H, A          ;保存结果
        ;
        ;其他代码段
        ;
        AJMP    0540H           ;绝对转移指令
        MOV     A, #5AH         ;寄存器 A 赋值
        RR      A               ;循环右移
        MOV     21H, A          ;保存结果

        ORG     0540H
Next:   MOV     PSW, #80H        ;CY 值 1
        MOV     A, #5AH         ;寄存器 A 赋值
        RLC     A               ;带进位循环左移
        MOV     22H, A          ;保存结果
        MOV     PSW, #80H        ;CY 值 1
        MOV     A, #5AH         ;寄存器 A 赋值
        RRC     A               ;带进位循环右移
        MOV     23H, A          ;保存结果
        LJMP    START
        END
```

在该程序中，AJMP 指令处程序指针为 PC=0x0105。执行跳转时，先将 PC 值加 2，然后取其高 5 位及 0x0540 的低 11 位组合而成 0x0540H 来替换 PC 值，即程序跳转到 Next 标号位置处执行。

!

注意：被替换的 PC 值是本条指令地址加 2 以后的 PC 值，即指向下一条指令的 PC 值，即 PC 当前值。

3. 短转移指令

短转移指令是一种相对寻址方式转移指令，指令的格式如下：

```
SJMP    rel
```

其中，rel 为相对偏移量，是一个带符号的 8 位二进制补码数，其范围为-128~+127。指令用 rel 来计算目的地址，并按计算得到的目的地址实现程序的相对转移。如 rel 为正数则表示向

前转移；如 **rel** 为负数则表示向后转移。转移的范围是 256，因此称为“短转移”。

该指令给出的是相对转移地址，而不是具体地址，这样当程序地址发生变化时，只要相对地址不变，该指令就不做任何改动。

短转移指令是 2 字节指令，其目的地址的计算公式如下：

目的地址 = (PC) + 2 + **rel**

对于短转移指令的使用可从如下两个方面进行讨论。

➤ 根据偏移量 **rel** 计算转移的目的地址

这种情况经常在读目标程序时遇到，是解决往哪儿转移的问题。

例如，在 835AH 地址上有 **SJMP** 指令如下：

835AH: SJMP 35H

源地址为 835AH，**rel**=35H 是正数，因此程序向前转移。

目的地址=835AH+35H=8931H。执行本指令后，程序转到 8931H 地址去执行。

例如，在 835AH 地址上 **SJMP** 指令如下：

835AH: SJMP 0E7H

源地址为 835AH，**rel**=0E7H 是负数 19H 的补码，因此程序向后转移。

目的地址=835AH+02H-19H=8343H。执行本指令后，程序向后转到 8343H 地址去执行。

➤ 根据目的地址计算偏移量

这是编程时必须解决的问题。对于 2 字节的 **SJMP** 指令，**rel** 的计算公式如下：

向前转移时，公式如下所示。

rel = 目的地址 - 源地址 - 2 = (地址差) - 2

向后转移时，目的地址小于源地址，**rel** 应为负数的补码。

rel = (目的地址 - (源地址 + 2))

= FFH - (源地址 + 2 - 目的地址) + 1

= FEH - (地址差)

此外，在汇编语言程序中，为等待中断或程序结束，常有使程序“原地踏步”的需要，对此可使用如下 **SJMP** 指令完成。

HERE: SJMP HERE

或者如下：

HERE: SJMP \$

指令机器码为 80FE。在汇编语言中，以“\$”代表 PC 的当前值。

在汇编程序中都有计算偏移量的功能。用户编写汇编源程序时，只需在相对转移指令中直接写上要转向的目的地址的地址标号就可以了。程序汇编时由汇编程序自动计算和填入偏移量。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:   MOV     A, #10H      ;立即寻址赋值
        MOV     R0, #20H     ;立即寻址赋值
        MOV     R1, #30H     ;立即寻址赋值
        SJMP 1000H          ;跳转
END
```

在该程序中，使用立即寻址完成赋值操作，最后使用相对寻址的跳转指令，跳转到 1000H 处循环执行。在 Keil μ Vision3 中进行编译，可以查看其机器码，从而推断偏移量的计算。

但在进行手动汇编时，偏移量的值则需由程序设计人员自己计算。示例如下：

```
LOOP : SJMP      LOOP1
```

假设 LOOP 的标号值为 0100H（即 SJMP 这条指令的机器码存于 0100H 和 0101H 两个单元之中），标号 LOOP1 的值为 0123H，即跳转的目标地址为 0123H，则指令的第二个字节（相对偏移量）应计算如下：

```
rel=0123H - 0102H=21H
```

4. 变址寻址转移指令

变址寻址转移指令是采用变址寻址方式的转移指令。它是单字节指令，指令的格式如下：

```
JMP      @A+DPTR      ; PC←( A ) + ( DPTR )
```

该指令以 DPTR 内容为基址，而以 A 的内容作为变址，将累加器 A 中的内容和基址寄存器 DPTR 的内容相加，其和作为转移的目的地址送入 PC，即目的地址=（A）+（DPTR）。指令执行后不改变 A 和 DPTR 的内容，也不影响任何标志位。

只要把 DPTR 中放入多分支转移的指令表首地址固定，而赋给 A 不同的值，就可实现程序的多分支转移。键盘译码程序就是本指令的一个典型应用。多分支程序的结构如下：

```
JMP      @A+DPTR
JPTAB:   AJMP FUN0
          AJMP FUN 1
          AJMP FUN 2
          AJMP FUN 3

FUN 0:
...
FUN 1:
...
FUN 2:
...
FUN 3:
...
```

下面通过给单片机的 4 个并行端口赋值，来讲解变址寻址转移指令在多分支程序中的应用。硬件采用前面的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:   MOV      P0,#00H      ; P0 端口赋值

        MOV      A,#00H
        MOV      DPTR,#1200H
        JMP      @A+DPTR      ; 变址寻址，跳转至 FUN0

TT1:    MOV      A,#02H
        MOV      DPTR,#1200H
        JMP      @A+DPTR      ; 变址寻址，跳转至 FUN1

TT2:    MOV      A,#04H
        MOV      DPTR,#1200H
        JMP      @A+DPTR      ; 变址寻址，跳转至 FUN2

ORG      1200H      ; 程序分支
AJMP FUN0
AJMP FUN1
AJMP FUN2

FUN0:   MOV      P1,#01H      ; 分支 0，FUN0
        LJMP TT1

FUN1:   MOV      P2,#02H      ; 分支 1，FUN1
```



```
LJMP TT2
FUN2:  MOV      P3, #03H      ;分支 2 , FUN2

LJMP START

END
```

在该程序中，定义了三个分支程序 FUN0、FUN1 和 FUN2。通过为累加器 A 和数据指针 DPTR 赋值，来实现不同分支的切换。程序在各个分支中分别对端口进行赋值操作。

将程序在 Keil μVision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0、P1、P2 和 P3 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μVision3 中进行单片机端口仿真，仿真结果如图 16-45 所示。

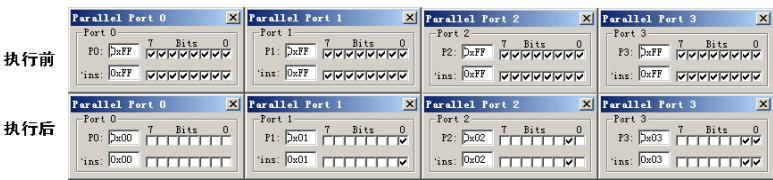


图 16-45 端口仿真结果

16.6.2 条件转移指令

条件转移是程序按照一定的条件进行转移的操作指令。执行条件转移指令时，如指令中规定的条件满足，则进行程序转移；否则程序顺序执行，相当于高级语言中的“if”语句。条件转移有如下几种。

1. 累加器判零转移指令

累加器判零转移指令是以累加器 A 的内容是否为零作为判断条件的转移指令。共有两个指令，这两个指令都是 2 字节指令，指令的格式如下：

```
JZ    rel      ;若 (A) = 0 , 则 PC← (PC) +2+rel
        ;若 (A) ≠0 , 则 PC← (PC) +2
JNZ   rel      ;若 (A) ≠0 , 则 PC← (PC) +2+rel
        ;若 (A) = 0 , 则 PC← (PC) +2
```

其中，JZ 指令的功能是当 (A) =0 时转移，否则按顺序向下执行；JNZ 指令的功能是当 (A) ≠ 0 时转移，否则顺序执行。这两个指令具有相反的功能。

在指令中，以 rel 为偏移量。在编写程序时，与短转移指令一样，常用目的地址的地址标号来代替 rel，由汇编程序自动换算成 8 位相对地址。

下面通过数据传送操作来讲解累加器判零转移指令在具体程序中的应用。程序中将外部数据 RAM 的一个数据块传送到内部数据 RAM 中，两者的首地址分别为 DATA1 和 TATA2，遇到传送的数据为 0 时停止。硬件采用前面的 AT89S52 的最小系统并配以外部 RAM 即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:   MOV      R0, #00H      ;外部数据块首地址
        MOV R1, #20H          ;内部数据块首地址
LOOP:    MOVX A, @R0           ;外部数据送给 A
HERE:    JZ       HERE         ;若=0 则终止
        ;若≠0 传送内部 RAM 数据
        MOV      @R1,A
```

```
INC      R0          ;修改地址指针
INC      R1
SJMP LOOP           ;继续循环
END
```

在该程序中，外部 RAM 向内部 RAM 的数据传送借助于累加器 A，利用累加器判零转移指令正好可以判别是否要继续传送或终止。

将程序在 Keil μ Vision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。当然也可以在 Keil μ Vision3 中，通过单步仿真操作来查看程序的执行顺序。

2. 数值比较转移指令

数值比较转移指令是将两个操作数进行比较，将比较结果作为条件来控制程序转移的操作指令。按照所采用的寻址方式，这类指令共有 4 条，其指令格式如下：

```
CJNE A,  #data,  rel      ; ( A ) =data , 则 PC← ( PC ) +3 , CY←0
                           ; ( A ) >data , 则 PC← ( PC ) +3+rel , CY←0
                           ; ( A ) <data , 则 PC← ( PC ) +3+rel , CY←1
CJNE A,  direct, rel      ; ( A ) =data , 则 PC← ( PC ) +3 , CY←0
                           ; ( A ) >data , 则 PC← ( PC ) +3+rel , CY←0
                           ; ( A ) <data , 则 PC← ( PC ) +3+rel , CY←1
CJNE Rn, # data,  rel      ; ( A ) =data , 则 PC← ( PC ) +3 , CY←0
                           ; ( A ) >data , 则 PC← ( PC ) +3+rel , CY←0
                           ; ( A ) <data , 则 PC← ( PC ) +3+rel , CY←1
CJNE @Ri, #data,  rel      ; ( A ) =data , 则 PC← ( PC ) +3 , CY←0
                           ; ( A ) >data , 则 PC← ( PC ) +3+rel , CY←0
                           ; ( A ) <data , 则 PC← ( PC ) +3+rel , CY←1
```

这 4 条指令的功能是将指令中的两个操作数进行比较，若相等，则程序顺序执行；若不相等，则程序转移。同时以进位标志位 CY 判断两个操作数大小，若目的操作数小于源操作数，则 (CY) =1，其他情况下 (CY) =0。

！注意：数值比较转移指令是三字节指令，因此 PC 当前值为 PC+3，转移目的地址为 PC+3+偏移量 rel。

数值比较转移指令是将两个操作数作为无符号数处理，因此 CJNE 指令不能直接用于有符号数大小比较。如要进行有符号数大小比较，则应根据符号位和进位标志位 CY 进行判别。

下面分别采用以上 4 种指令来讲解数值比较转移指令在具体程序中的应用。硬件采用前面的 AT89S52 最小系统即可，程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      A, #00H
          CJNE A, #12H, FUN0          ;立即寻址方式

TT1:      MOV      A, #02H
          MOV      20H, #20H
          CJNE A, 20H, FUN1          ;直接寻址方式

TT2:      MOV      A, #05H
          MOV      R0, #20H
          CJNE R0, #21H, FUN2        ;寄存器寻址方式

TT3:      MOV      A, #15H
```

```
MOV      21H, #21H
MOV      R1, 21H
CJNE @R1, #23H, FUN3      ; 寄存器间接寻址方式
```

```
FUN0:    MOV      P0, #01H      ; 分支 0, FUN0
         LJMP TT1
FUN1:    MOV      P1, #100      ; 分支 1, FUN1
         LJMP TT2
FUN2:    MOV      P2, #100B     ; 分支 2, FUN2
         LJMP TT3
FUN3:    MOV      P3, #100      ; 分支 3, FUN3

         LJMP START
         END
```

在该程序中，建立了 4 个分支程序用于对端口进行赋值。分别采用立即寻址、直接寻址、寄存器寻址和寄存器间接寻址方式来进行数值比较，如果数值不相等则跳转到对应的分支来执行，从而实现对单片机端口的赋值操作。

将程序在 Keil μ Vision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0、P1、P2 和 P3 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μ Vision3 中进行单片机端口仿真，仿真结果如图 16-46 所示。

3. 减 1 条件转移指令

减 1 条件转移指令是首先对源操作数执行减 1 操作，然后判断结果是否为零，根据结果进行跳转的操作指令。减 1 条件转移指令共有以下两条指令。

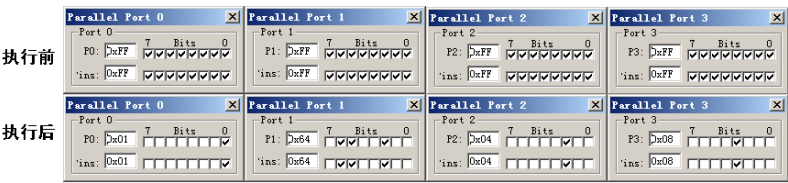


图 16-46 端口仿真结果

➤ 寄存器减 1 条件转移指令

```
DJNZ Rn,      rel      ; Rn ← ( Rn ) -1
                        ; ( Rn ) ≠ 0, 则 PC← ( PC ) +2+rel
                        ; ( Rn ) = 0, 则 PC← ( PC ) +2
```

该指令的功能是寄存器内容减 1，如果所得结果为 0，则程序顺序执行；如所得结果不为 0，则程序转移。寄存器减 1 条件转移指令是 2 字节指令。

➤ 直接寻址单元减 1 条件转移指令

```
DJNZ direct, rel      ; direct ← ( direct ) -1
                        ; ( direct ) ≠ 0, 则 PC← ( PC ) +3+rel
                        ; ( direct ) = 0, 则 PC← ( PC ) +3
```

该指令的功能是直接寻址单元内容减 1，如果所得结果为 0，则程序顺序执行；如所得结果不为 0，则程序转移。直接寻址单元减 1 条件转移指令是 3 字节指令。

这两条指令主要用于控制程序循环。例如将寄存器或内部 RAM 单元赋值循环次数作为计数器，利用减 1 条件转移指令，以减 1 后是否为 0 作为转移条件，即可实现按次数控制循环。

下面通过具体的实例来讲解 DJNZ 指令在具体的程序中的应用。程序示例如下：

```
ORG      0000H
JMP      START
```

51 单片机开发与应用技术详解

```

                                ORG      1000H
START:  MOV      DPTR, #2000H      ;源数据区首地址
                                PUSH     DPL      ;源首址暂存堆栈
                                PUSH     DPH
                                MOV      DPTR, #3000H      ;目的数据区首址
                                MOV      R2,   DPL      ;目的首址暂存寄存器
                                MOV      R3,   DPH
LOOP:   POP      DPH      ;取回源地址
                                POP      DPL
                                MOVX     A, @DPTR      ;取出数据
                                INC      DPTR      ;源地址增量
                                PUSH DPL      ;源地址暂存堆栈
                                PUSH DPH
                                MOV      DPL, R2      ;取回目的地址
                                MOV      DPH, R3
                                MOVX     @DPTR, A      ;数据送目的区
                                INC      DPTR      ;目的地址增量
                                MOV      R2,   DPL      ;目的地址暂存寄存器
                                MOV      R3,   DPH
                                DJNZ 35H, LOOP      ;继续循环
                                RET      ;返回主程序
                                END
```

在该程序中，把 2000H 开始的外部 RAM 单元中的数据送到 3000H 开始的外部 RAM 单元中，数据个数已在内部 RAM 的 35H 单元中。通过对 35H 单元进行减 1 来进行条件转移操作。

从上述两种转移指令可以看出，无条件转移指令有直接寻址和相对寻址两种方式，而条件转移指令则只有相对寻址一种寻址方式。

以上这些条件转移指令都是相对转移指令，转移范围有限 256B (-128~+127)，若要在大范围实现条件转移，可以将条件转移指令和长转移指令 LJMP 结合使用。

例如，根据累加器 A 和立即数 80H 比较的结果，转移到标号 NEXT1，其转移距离已超过 256B。

```

                                CJNE A, #80H, NEXT      ;不相等，则转移
                                ...                    ;相等，则顺序执行
                                SJMP NEXT2      ;执行完毕，转移到 NEXT2
NEXT:   LJMP NEXT1      ;长转移至 NEXT1
```

将 CJNE 和 LJMP 指令结合，可以实现 64KB 范围内的条件转移。其中“SJMP NEXT2”指令是在执行完两数相等处理后，转移到顺序执行的位置，以免造成混乱。

16.6.3 子程序调用及返回指令

子程序是在一个主程序中被调用的程序段。在一个程序中经常遇到反复多次执行某程序段的情况，如果重复编写这个程序段，会使程序变得冗长而杂乱。为此可采用“子程序结构”，即将重复的程序段编写为一个子程序，通过主程序调用而使用，这样不但减少了编程工作量，而且也缩短了程序的长度。子程序结构是一种重要的程序结构。

调用子程序的程序称为主程序，主程序和子程序之间的调用关系如图 16-47 所示。

如果在子程序中还有其他子程序，称为子程序嵌套，如图 16-48 所示。子程序调用指令将程序中断位置的地址在堆栈中保存起来，堆栈的先入后出的存取方式适合存放断点地址，尤其在有子程序嵌套的情况下。

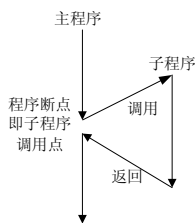


图 16-47 子程序调用示意图

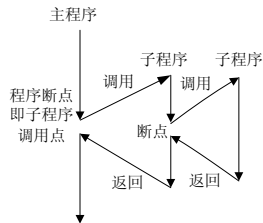


图 16-48 子程序的嵌套调用

调用和返回构成了子程序调用的完整过程。为了实现这一过程，必须有子程序调用指令和返回指令。调用指令在主程序中使用，而返回指令则应该是子程序的最后一条指令。执行完这条指令之后，程序返回主程序断点处继续执行。

51 系列单片机有如下两条子程序调用指令和一条返回指令。

1. 绝对调用指令

绝对调用指令是采用 ACALL 的子程序调用指令。其中 ACALL 是 2 字节指令，其指令格式如下：

```
ACALL    addr11                                ; PC ← ( PC ) + 2
                                                ; SP ← ( SP ) + 1 , ( SP ) ← ( PC ) 7~0
                                                ; SP ← ( SP ) + 1 , ( SP ) ← ( PC ) 15~8
                                                ; PC10~0 ← addr11
```

在指令的操作数提供了子程序入口地址的低 11 位，这 11 位地址的 $A_7 \sim A_0$ 在指令的第二字节中， $A_{10} \sim A_8$ 则占据第一字节的高三位。指令的操作码 10001 占据第一字节的低 5 位。

该指令的功能是先将 PC 加 2，指向下一条指令地址（断点地址），将断点地址压入堆栈，待子程序返回时再送回 PC，再以指令提供的子程序入口 11 位地址取代 PC 的低 11 位，而 PC 的高 5 位不变。使程序转移到对应的子程序入口处执行。

该指令给出了子程序入口地址的低 11 位，因此绝对调用指令的子程序调用范围是 2KB。

示例如下：

```
8100H:  ACALL    48H
```

Addr11 的高三位 $A_{10} \sim A_8 = 100$ ，因此指令第一字节为 91H，第二字节为 8FH，构造的目的地址 PC 加 2 后 $(PC) = 8102H$ ，即 1000000100000010B (8102H)。指令提供的 11 位地址是 10010001111B。替换 PC 的低 11 位后，PC 的值即形成的目的地址为 1000010010001111B (848FH)，即被调用子程序入口地址为 848FH，或者说主程序到 848FH 处去调用子程序。

本调用指令的地址为 8100H，不变的高 5 位是 10000B，因此本指令的子程序调用范围是 8000H~87FFH。下面采用 ACALL 指令来讲解绝对调用指令在具体程序中的应用，程序示例如下：

```
ORG      0000H
JMP      START
ORG      0100H
START:   MOV     A, #5AH          ; 寄存器 A 赋值
        RL      A                ; 循环左移
        MOV     20H, A           ; 保存结果
        ;
        ; 其他代码段
        ;
        ACALL   0540H            ; 绝对调用指令
        MOV     A, #5AH          ; 寄存器 A 赋值
        RR      A                ; 循环右移
        MOV     21H, A           ; 保存结果
```

```

                                LJMP START

                                ORG      0540H
Next:  MOV      PSW, #80H        ;CY 值 1
        MOV      A, #5AH        ;寄存器 A 赋值
        RLC      A              ;带进位循环左移
        MOV      22H, A         ;保存结果
        MOV      PSW, #80H      ;CY 值 1
        MOV      A, #5AH        ;寄存器 A 赋值
        RRC      A              ;带进位循环右移
        MOV      23H, A         ;保存结果
        RET

                                END
```

在该程序中，定义了子程序 Next。ACALL 指令处程序指针为 PC=0x0105。执行跳转时，先将 PC 值加 2，然后取其高 5 位及 0x0540 的低 11 位组合而成 0x0540H 来替换 PC 值，即程序跳转到 Next 标号位置处的子程序继续执行。

2. 长调用指令

长调用指令是采用 LCALL 的子程序调用指令。其中 LCALL 是 3 字节指令，其指令格式如下：

```

LCALL    addr16                ;PC←( PC ) +3
                                ;SP←( SP ) +1 , ( SP ) ← ( PC ) 7~0
                                ;SP←( SP ) +1 , ( SP ) ← ( PC ) 15~8
                                ;PC15~0←addr16
```

在长调用指令中，调用地址由指令的操作数直接给出。

该指令的功能是先将 PC 加 3，指向下一条指令地址（断点地址），再将断点地址压入堆栈保存，然后以指令中的 addr16 作为地址调用子程序入口地址装入 PC，使程序转移到子程序入口处执行。

长调用指令的子程序调用范围是 64KB。Addr16 就是被调用子程序的入口地址，使用比较方便，但 3 字节指令较长，占用存储空间较多。

例如，(SP)=60H，标号地址 START 为 0100H，标号 MIR 为 8100H，执行如下指令。

```

START:  LCALL    MIR
```

结果为 (SP)=62H，(61H)=03H，(62H)=01H，(PC)=8100H。

下面采用 LCALL 指令来讲解长调用指令在具体程序中的应用，程序示例如下：

```

                                ORG      0000H
                                JMP      START
                                ORG      0100H
START:  MOV      A, #7AH        ;寄存器 A 赋值
        RL      A              ;循环左移
        MOV      20H, A        ;保存结果
        ;
        ;其他代码段
        ;
        LCALL    Next          ;长调用
        MOV      A, #5AH        ;寄存器 A 赋值
        RR      A              ;循环右移
        MOV      21H, A        ;保存结果
```

```
LJMP START

Next:  ORG      0540H
        MOV     PSW, #80H      ;CY 值 1
        MOV     A, #18H        ;寄存器 A 赋值
        RLC     A              ;带进位循环左移
        MOV     22H, A          ;保存结果
        MOV     PSW, #80H      ;CY 值 1
        MOV     A, #5AH        ;寄存器 A 赋值
        RRC     A              ;带进位循环右移
        MOV     23H, A          ;保存结果
        RET

END
```

在该程序中，定义了子程序 Next。长调用指令 LCALL 处直接给出子程序的标号地址，即程序跳转到 Next 标号位置处的子程序继续执行。

3. 返回指令

返回指令是在子程序末尾，用于返回主程序继续执行的操作指令。返回指令共有以下两条。

➤ 子程序返回指令

```
RET                                     ;PC15~8←( ( SP ) ) , SP←( SP )-1
                                       ;PC7~0←( ( SP ) ) , SP←( SP )-1
```

子程序返回指令的功能是执行子程序返回，从堆栈中自动取出断点地址送给程序计数器 PC，使程序在主程序断点处顺序向下执行，该指令放在子程序的末尾。

例如，(SP)=62H，(62H)=07H，(61H)=30H。

```
RET
执行指令的结果为 (SP)=60H，(PC)=0730H，CPU 从 0730H 开始执行程序。
```

➤ 中断子程序返回指令

```
RETI                                  ;PC15~8←( ( SP ) ) , SP←( SP )-1
                                       ;PC7~0←( ( SP ) ) , SP←( SP )-1
```

中断子程序返回指令，除具有子程序返回指令所具有的功能外，同时还具有清除中断响应时被置位的优先级状态触发器、开放较低级中断，并恢复中断逻辑以接收新的中断请求等功能。

使用返回指令时，要注意 RET 和 RETI 不能互换使用，且在子程序或中断服务子程序中，PUSH 和 POP 指令必须成对使用，否则无法正确返回主程序的断点位置。

16.7 位操作指令

位操作指令是对位变量寄存器等进行按位处理的指令。其中位处理，就是以位 (bit) 为单位进行的运算和操作。位变量也称为布尔变量或开关变量。51 系列单片机具有丰富的布尔变量处理功能，即位处理功能。

51 系列单片机可以把由模拟电路或数字电路实现的控制功能改由软件方法实现，使控制系统软化的微控制技术，以及位处理器的相关内容都与位操作指令有关。

51 系列单片机位处理器的硬件资源包括如下几种。

- 运算器中的 ALU，与字节处理合用；
- 程序存储器，与字节处理合用；

- 位累加器 **CY**，是位传送的中心。在字节处理中有一个累加器 **A**，而位处理需要使用进位标志 **CY** 作为累加器，为了便于区分，把 **A** 称之为字节累加器，而把 **CY** 称为位累加器，在指令中用 **C** 表示，以便于书写；
- 内部 **RAM** 位寻址区的 128 个可寻址位；
- 专用寄存器中的可寻址位；
- **I/O** 口的可寻址位。

位操作指令是位处理器专门处理布尔变量的软件资源，位操作指令是 51 系列单片机指令系统的指令子集，用以进行位的传送、置位、清零、取反、位状态判跳、位逻辑运算、位输入与输出等位操作。

位操作指令的操作对象有内部 **RAM** 中的位寻址区，即 20H~21H 中的 128 位(位地址 00H~7FH)，以及特殊功能寄存器中可以寻址的各位。

位地址在指令中用 **bit** 表示，**bit** 有 4 种表示形式，分别是采用直接地址、字节地址加位序号、位名称，或者特殊功能寄存器加位序号表示。位操作指令共有 17 条。

16.7.1 位变量传送指令

位变量传送指令是在可寻址位与位累加器 **CY** 之间传送数据的操作指令。位变量传送指令一般不影响其他标志位。按照传送方向不同，共有两条指令，格式分别如下：

```
MOV      C,  bit                ;CY←( bit )
MOV      bit, C                ;bit←( CY )
```

在 51 系列单片机指令系统中，没有两个可寻址位之间的传送指令，无法实现直接传送。如需要进行这种传送，应使用这两条指令，以 **CY** 作中介来实现。

下面通过将 0AH 位的内容传送到 3AH 位的操作，来讲解位变量传送指令在具体程序中的应用。硬件采用前面的 AT89S52 最小系统即可，程序示例如下：

```
ORG      0000H
JMP      START
ORG      0100H
START:   MOV      21H, #0FFH      ;RAM 21H 单元赋值
MOV      10H, C                  ;暂存 CY 内容
MOV      C, 0AH                  ;0AH 位送 CY
MOV      3AH, C                  ;CY 送 3AH 位
MOV      C, 10H                  ;恢复 CY 内容
RET
END
```

在该程序中，首先为片内 **RAM** 的 21H 单元赋值，然后将其中的 0AH 位传送给 **CY**。接着，将 **CY** 的值传送给 3AH 位，并将原来 **CY** 的值恢复。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 **RAM** 的保存结果。程序仿真结果如图 16-49 所示。

注意：位指令中的地址都是位地址，而不是存储单元地址。

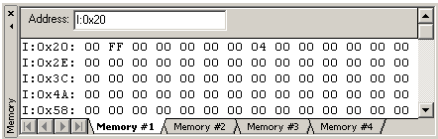


图 16-49 程序仿真结果

16.7.2 置位与清零指令

置位与清零指令是对 CY 及可寻址位进行置位或清零的操作指令，共有 4 条指令，格式如下：

```
SETB    C                                ; CY←1
SETB    bit                               ; bit←1
CLR     C                                ; CY←0
CLR     bit                               ; bit←0
```

这些操作指令不影响其他标志位。

下面分别采用上述的这几种方式来讲解置位与清零指令在程序中的应用，硬件采用前面介绍的 AT89S52 的最小系统即可。程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H

START:   SETB 20H.0                      ;置 20H.0 为 1
        SETB 01H                        ;置 20H.1 为 1
        CLR  02H                        ;置 20H.2 为 0
        SETB 03H                        ;置 20H.3 为 1
        CLR  20H.4                      ;置 20H.4 为 0
        MOV  P0,20H                    ;将片内 RAM 的 20H 单元数据输出到 P0 端口
        MOV  P1,00H                    ;P1 端口清零
        SETB P1.0                      ;置 P1.0 为 1
        SETB RS0                       ;特殊功能寄存器 PSW 的 D2 位
        SETB PSW.3                     ;特殊功能寄存器 PSW 的 D3 位
        JMP  START                    ;跳转
END
```

在该程序中，首先为片内 RAM 的 20H 单元采用置位与清零指令赋值，然后将 20H 单元的数据输出到 P0 端口。接着对 P1.0 端口进行位寻址赋值。最后，采用两种方法对特殊功能寄存器的 D2 和 D3 位赋值。

将程序在 Keil μVision3 中进行编译，输出的可执行文件下载到硬件电路中，便可以运行。分别测量 P0 和 P1 端口的电平，可以验证程序的执行结果。当然也可以在 Keil μVision3 中进行单片机端口仿真，仿真结果如图 16-50 所示。

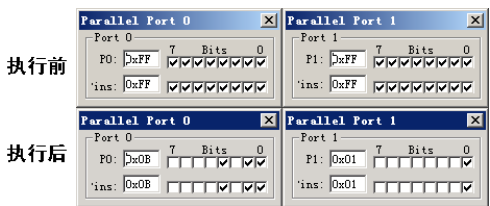


图 16-50 端口仿真结果

16.7.3 位逻辑运算指令

位逻辑运算指令是进行逻辑运算的操作指令。位逻辑运算指令包括与、或、非三种，共 6 条指令。指令的格式如下：

```
ANL     C,  bit                        ; CY←( CY ) ∧ ( bit )
ANL     C,  /bit                       ; CY←( CY ) ∧ (  $\overline{\text{bit}}$  )
ORL     C,  bit                        ; CY←( CY ) ∨ ( bit )
```

```
ORL      C,    /bit                ; CY←( CY ) ∨ (  $\overline{\text{bit}}$  )
CPL      C                                ; CY←(  $\overline{\text{CY}}$  )
CPL      bit                            ; bit←(  $\overline{\text{bit}}$  )
```

ANL 和 ORL 分别是将位累加器 CY 的内容与位地址中的内容（或取反后的内容）进行与、或操作，结果送入 CY 中。“/”表示将该位值先求反后再运算，但是不改变位地址中原来的值。指令 CPL 是将位累加器 CY 或位地址中的内容取反。

在位操作指令中，没有提供位的异或运算，但是异或运算可以通过以上位逻辑运算来组合得到。例如，假设 X、Y、Z 代表位地址，进行 X、Y 内容的异或操作，结果送 Z。可以将异或转化，公式为 $Z=X \oplus Y=\overline{X}Y+X\overline{Y}$ 。

此外，通过位逻辑运算还可以对各种组合逻辑电路进行模拟，即在单片机中，用软件方法来获得组合电路的逻辑功能。

多数位操作指令与字节的位操作操作指令的助记符完全相同，易发生混淆。位操作指令中有“C”作为操作数，可以以此区别；而和“CLR bit”和“CPL bit”两条指令相对应的只有“CLR A”和“CPL A”；SETB 指令是位操作独有的。

下面分别采用上述 6 个指令来讲解位逻辑运算指令在具体程序中的应用，程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      20H,#0FFH        ;片内 RAM 的 20H 单元赋值
          MOV      C,1              ;CY 置 1
          ANL      C,20H.0          ;位逻辑与运算
          MOV      2EH.0,C          ;保存运算结果
          ANL      C,/20H.1         ;先取反再位逻辑与运算
          MOV      2EH.1,C          ;保存运算结果

          ORL      C,20H.2          ;位逻辑或运算
          MOV      2EH.2,C          ;保存运算结果
          ORL      C,/20H.3         ;先取反再位逻辑或运算
          MOV      2EH.3,C          ;保存运算结果

          CPL      2EH.4            ;取反运算
          CPL      C                ;取反运算
          MOV      2EH.5,C          ;保存运算结果

          JMP      START            ;跳转
END
```

在该程序中，首先为所使用到的寄存器及 RAM 单元赋值，然后分别采用上述 6 种位逻辑运算指令进行运算。运算的结果保存在片内 RAM 的 2EH 单元。

将程序在 Keil μ Vision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-51 所示。

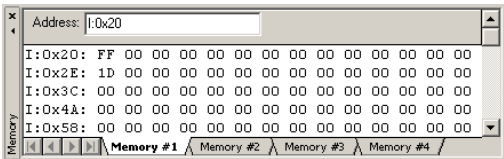


图 16-51 程序仿真结果

16.7.4 位控制转移指令

位控制转移指令是以位的状态（CY 或 bit）作为程序转移判断条件的条件转移指令。位控制转移指令分为两类，共有以下 5 条。

1. 以 CY 状态为条件的转移指令

以 CY 状态为条件的转移指令是以进位位 CY 的状态作为转移判断条件的操作指令。其中，若 CY 为 1，则转移；否则顺序执行。有两条指令，格式分别如下：

```
JC      rel                ; 若 (CY) =1, 则 PC←(PC)+2+rel
                        ; 若 (CY) ≠1, 则 PC←(PC)+2
JNC     rel                ; 若 (CY) =1, 则 PC←(PC)+2+rel
                        ; 若 (CY) ≠1, 则 PC←(PC)+2
```

以 CY 状态为条件的转移指令均为 2 字节指令。

下面分别采用以上两条指令，来讲解以 CY 状态为条件的转移指令在具体程序设计中的应用。硬件采用前面的 AT89S52 最小系统即可，程序示例如下：

```
ORG      0000H
JMP      START
ORG      1000H
START:    MOV      PSW, #80H      ; 赋 CY=1
JC        FUN1      ; 若 CY=1, 则跳转
FF1:      MOV      PSW, #00H      ; 赋 CY=0
JNC       FUN2      ; 若 CY=0, 则跳转

FUN1:     MOV      20H, #0ABH     ; 片内 RAM 的 20H 单元赋值
JMP       FF1
FUN2:     MOV      21H, #0CDH     ; 片内 RAM 的 21H 单元赋值

RET
END
```

在该程序中，首先为 PSW 的进位位 CY 赋值，然后分别采用 JC 和 JNC 指令来完成条件转移。在程序中，在 FUN1 和 FUN2 处分别为片内 RAM 的 20H 单元和 21H 单元赋值。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-52 所示。

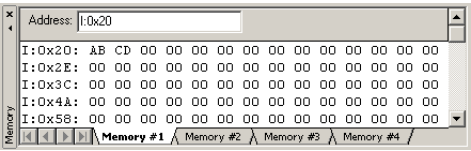


图 16-52 程序仿真结果

2. 以位状态为条件的转移指令

以位状态为条件的转移指令是以直接寻址位的状态为判断转移条件的操作指令。其中，若 bit 为 1，则转移；否则顺序执行。共有三条指令，格式如下：

```
JB      bit, rel          ; 若 (bit) =1, 则 PC←(PC)+3+rel
                        ; 若 (bit) ≠1, 则 PC←(PC)+3
JNB     bit, rel          ; 若 (bit) =0, 则 PC←(PC)+3+rel
                        ; 若 (bit) ≠0, 则 PC←(PC)+3
JBC     bit, rel          ; 若 (bit) =1, 则 PC←(PC)+3+rel, 且 bit←0
```

；若 (bit) ≠1，则 PC← (PC) +3

以位状态为条件的转移指令均为 3 字节指令。

下面分别采用以上三条指令，来讲解以位状态为条件的转移指令在具体程序设计中的应用。硬件采用前面的 AT89S52 最小系统即可，程序示例如下：

```

      ORG      0000H
      JMP      START
      ORG      1000H

START:  MOV     20H,#0FFH      ;赋 RAM 的 20H 单元为 0xFF
      JB       20H.0,FUN1     ;若 bit=1，则跳转

FF1:    MOV     21H,#00H      ;赋值
      JNB      21H.0,FUN2     ;若 bit=0，则跳转

FF2:    MOV     22H,#80H      ;赋值
      JBC      22H.7,FUN3     ;跳转

FUN1:   MOV     30H,#0ABH     ;片内 RAM 的 30H 单元赋值
      JMP      FF1

FUN2:   MOV     31H,#0CDH     ;片内 RAM 的 31H 单元赋值
      JMP      FF2

FUN3:   MOV     32H,#0EFH     ;片内 RAM 的 32H 单元赋值

      RET
      END
```

在该程序中，首先为位寻址区 20H、21H 和 22H 单元赋值，然后分别采用 JB、JNB 和 JBC 指令来完成条件转移。程序中，在 FUN1、FUN2 和 FUN3 处分别为片内 RAM 的 30H 单元、31H 和 32H 单元赋值。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，寄存器中数值的变化及片内 RAM 的保存结果。程序仿真结果如图 16-53 所示。

！注意：JB 和 JBC 指令，两者的转移条件相同，所不同的是 JBC 指令在转移的同时，还将直接寻址位清零。

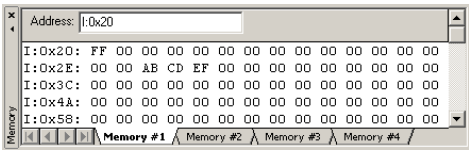


图 16-53 程序仿真结果

16.8 空操作指令

空操作指令是指不进行任何操作的指令。51 系列单片机中提供了空操作指令，其格式如下：

```

NOP                                     ; PC← ( PC ) +1
```

空操作指令控制 CPU 不作任何操作，只消耗一个机器周期的时间。空操作指令是单字节指令，因此执行后 PC 加 1，时间延续一个机器周期。利用 NOP 指令可以使程序进行精确的延时。

使用位操作指令在某些时候可以使程序设计更方便灵活，比用字节的逻辑运算要简洁、易读。例如，在 8051 的 P1.0 位输出一个方波，方波周期为 8 个机器周期。程序示例如下：

```

ORG      0000H
JMP      START
ORG      1000H
```

```
START:  SETB P1.0          ;使 P1.0 位输出电平“1”
        NOP
        NOP
        NOP                ;延时 3 个机器周期
        CLR      P1.0      ;使 P1.0 位输出电平“0”
        NOP
        NOP
        NOP                ;延时 3 个机器周期
        SETB P1.0          ;使 P1.0 位输出电平“1”
        SJMP $            ;暂停

        END
```

在该程序中，由于位操作指令占用一个机器周期，所以，程序中只用三个 NOP 语句执行三个机器周期的延时，再加上一个位操作的延时，便可以获得 4 个机器周期的高电平和 4 个机器周期的低电平。

将程序在 Keil μVision3 中进行编译，通过单步仿真调试，可以查看程序在每一步指令执行后，波形仿真界面的波形结果。程序仿真结果如图 16-54 所示。



图 16-54 波形仿真结果

16.9 51 系列单片机指令汇总

前面详细介绍了 51 系列单片机的各种指令，下面给出所有这些指令的汇总表，如表 16-2 所示。其中列出了指令的助记符、十六进制代码、执行的功能、占用的字节个数、机器周期个数和对各个标志位的影响等。

表 16-2 51 系列单片机指令汇总

指令类型	助 记 符	十六进制代码	功 能	占用字节个数	机器周期数	对标志位的影响			
						P	OV	AC	CY
数据传送类指令	MOV A,Rn	E8~EF	执行(A)←(Rn)，(n=0~7)	1	1	√	×	×	×
	XCH A,Rn	C8~CF	执行(A)↔(Rn)，(n=0~7)	1	1	√	×	×	×
	XCH A,@Ri	C6,C7	执行(A)↔((Ri))，(i=0~1)	1	1	√	×	×	×
	MOV A,@Ri	E6,E7	执行(A)←((Ri))，(i=0~1)	1	1	√	×	×	×
	XCHD A,@Ri	D6,D7	执行(A)3~0↔((Ri))3~0，(i=0~1)	1	1	√	×	×	×
	MOV @Ri,A	F6,F7	执行((Ri))←(A)，(i=0~1)	1	1	×	×	×	×
	MOV Rn,A	F8~FF	执行(Rn)←(A)，(n=0~7)	1	1	×	×	×	×

	MOVC A,@A+DPTR	93	执行(A)←((A)+(DPTR))	1	2	√	×	×	×
	MOVC A,@A+PC	83	执行(A)←((A)+(PC))	1	2	√	×	×	×
	MOVX A,@Ri	E2,E3	执行(A)←((P2)+(Ri))，(i=0-1)	1	2	√	×	×	×
	MOVX A,@DPTR	E0	执行(A)←((DPTR))	1	2	√	×	×	×
	MOVX @Ri,A	F2,F3	执行((Ri)+(P2))←(A)，(i=0-1)	1	2	×	×	×	×
	MOVX @DPTR,A	F0	执行((DPTR))←(A)	1	2	×	×	×	×
	MOV Rn,direct	A8~AF	执行(Rn)←(direct)，(n=0-7)	2	2	×	×	×	×
	MOV Rn,#data	78~7F	执行(Rn)←data，(n=0-7)	2	1	×	×	×	×
	MOV direct,A	F5	执行(direct)←(A)	2	1	×	×	×	×
	MOV direct,Rn	88~8F	执行(direct)←(Rn)	2	2	×	×	×	×
	MOV direct1,direct2	85	执行(direct1)←(direct2)	3	2	×	×	×	×
	MOV direct,@Ri	86,87	执行(direct)←((Ri))，(i=0-1)	2	2	×	×	×	×
	MOV direct,#data	75	执行(direct)←data	3	2	×	×	×	×
	MOV @Ri,direct	A6,A7	执行((Ri))←(direct)	2	2	×	×	×	×
	MOV @Ri,#data	76,77	执行((Ri))←data	2	1	×	×	×	×
	MOV DPTR,#data	90	执行(DPTR)←data	3	2	×	×	×	×
	MOV C,bit	A2	执行(CY)←(bit)	2	1	×	×	×	√
	MOV bit,C	92	执行(bit)←(CY)	2	2	×	×	×	×
	PUSH direct	C0	执行(SP)←(SP)+1，((SP))← (direct)	2	2	×	×	×	×

续表

指令 类型	助 记 符	十六 进制 代码	功 能	占用 字节 个数	机器 周期 数	对标志位的影响			
数据 传送类 指令	POP direct	D0	执行(direct)←((SP))，(SP)← (SP)-1	2	2	×	×	×	×
	MOV A,direct	E5	执行(A)←(direct)	2	1	√	×	×	×
	XCH A,direct	C5	执行(A)↔(direct)	2	1	√	×	×	×
	MOV A,#data	74	执行(A)←data	2	1	√	×	×	×
	ADD A,Rn	28~2F	执行(A)←(A)+(Rn)，(n=0-7)	1	1	√	√	√	√
	ADD A,direct	25	执行(A)←(A)+(direct)	2	1	√	√	√	√
	ADD A,@Ri	26,27	执行(A)←(A)+((Ri))，(i=0-1)	1	1	√	√	√	√
	ADD A,#data	24	执行(A)←(A)+data	2	1	√	√	√	√
	ADDC A,Rn	38~3F	执行(A)←(A)+(Rn)+(CY)，(n=0-7)	1	1	√	√	√	√
	ADDC A,direct	35	执行(A)←(A)+(direct)+(CY)	2	1	√	√	√	√
	ADDC A,@Ri	36,37	执行(A)←(A)+((Ri))+ (CY)，(i=0-1)	1	1	√	√	√	√
	ADDC A,#data	34	执行(A)←(A)+data+ (CY)	2	1	√	√	√	√
算 术 运 算 类 指 令	SUBB A,Rn	98~9F	执行(A)←(A)-(Rn)-(CY)，(n=0-7)	1	1	√	√	√	√
	SUBB A,direct	95	执行(A)←(A)-(direct)-(CY)	2	1	√	√	√	√
	SUBB A,@Ri	96,97	执行(A)←(A)-((Ri))-(CY)	1	1	√	√	√	√
	SUBB A,#data	94	执行(A)←(A)-data-(CY)	2	1	√	√	√	√
	INC A	04	执行(A)←(A)+1	1	1	√	×	×	×
	INC Rn	08~0F	执行(Rn)←(Rn)+1，(n=0-7)	1	1	×	×	×	×
	INC direct	05	执行(direct)←(direct)+1	2	1	×	×	×	×
	INC @Ri	06,07	执行((Ri))←((Ri))+1，(i=0-1)	1	1	×	×	×	×

	INC DPTR	A3	执行 $(DPTR) \leftarrow (DPTR) + 1$	1	2	×	×	×	×
	DEC A	14	执行 $(A) \leftarrow (A) - 1$	1	1	√	×	×	×
	DEC Rn	18~1F	执行 $(Rn) \leftarrow (Rn) - 1, (n=0-7)$	1	1	×	×	×	×
	DEC direct	15	执行 $(direct) \leftarrow (direct) - 1$	2	1	×	×	×	×
	DEC @Ri	16,17	执行 $((Ri)) \leftarrow ((Ri)) - 1, (i=0-1)$	1	1	×	×	×	×
	MUL AB	A4	执行 $(A)(B) \leftarrow (A) \times (B)$	1	4	√	√	×	√
	DIV AB	84	执行 $(A)(B) \leftarrow (A) \div (B)$	1	4	√	√	×	√
	DA A	D4	执行对 (A) 进行十进制调整	1	1	√	√	√	√
	ANL A,Rn	58~5F	执行 $(A) \leftarrow (A) \wedge (Rn), (n=0-7)$	1	1	√	×	×	×
	ANL A,direct	55	执行 $(A) \leftarrow (A) \wedge (direct)$	2	1	√	×	×	×
	ANL A,@Ri	56,57	执行 $(A) \leftarrow (A) \wedge ((Ri)), (i=0-1)$	1	1	√	×	×	×
	ANL A,#data	54	执行 $(A) \leftarrow (A) \wedge data$	2	1	√	×	×	×
	ANL direct,A	52	执行 $(direct) \leftarrow (direct) \wedge (A)$	2	1	×	×	×	×
	ANL direct,#data	53	执行 $(direct) \leftarrow (direct) \wedge data$	3	2	×	×	×	×
	ORL A,Rn	48~4F	执行 $(A) \leftarrow (A) \vee (Rn), (n=0-7)$	1	1	√	×	×	×
	ORL A,direct	45	执行 $(A) \leftarrow (A) \vee (direct)$	2	1	√	×	×	×

续表

指令类型	助记符	十六进制代码	功能	占用字节个数	机器周期数	对标志位的影响			
逻辑运算类指令	ORL A,@Ri	46,47	执行 $(A) \leftarrow (A) \vee ((Ri)), (i=0-1)$	1	1	√	×	×	×
	ORL A,#data	44	执行 $(A) \leftarrow (A) \vee data$	2	1	√	×	×	×
	ORL direct,A	42	执行 $(direct) \leftarrow (direct) \vee (A)$	2	1	×	×	×	×
	ORL direct,#data	43	执行 $(direct) \leftarrow (direct) \vee data$	3	2	×	×	×	×
	XRL A,Rn	68~6F	执行 $(A) \leftarrow (A) \oplus (Rn), (n=0-7)$	1	1	√	×	×	×
	XRL A,direct	65	执行 $(A) \leftarrow (A) \oplus (direct)$	2	1	√	×	×	×
	XRL A,@Ri	66,67	执行 $(A) \leftarrow (A) \oplus ((Ri)), (i=0-1)$	1	1	√	×	×	×
	XRL A,#data	64	执行 $(A) \leftarrow (A) \oplus data$	2	1	√	×	×	×
	XRL direct,A	62	执行 $(direct) \leftarrow (direct) \oplus (A)$	2	1	×	×	×	×
	XRL direct,#data	63	执行 $(direct) \leftarrow (direct) \oplus data$	3	2	×	×	×	×
	CLR A	E4	执行 $(A) \leftarrow 0$	1	1	√	×	×	×
	CPL A	F4	执行 $(A) \leftarrow (\overline{A})$	1	1	×	×	×	×
	RL	23	循环将 (A) 左移一位	1	1	×	×	×	×
	RLC A	33	循环将 (A), (CY) 左移一位	1	1	√	×	×	√
	RR A	03	循环将 (A) 右移一位	1	1	×	×	×	×
	RRC A	13	循环将 (A), (CY) 右移一位	1	1	√	×	×	√
	SWAP A	C4	将 (A) 半字节交换	1	1	×	×	×	×
	CLR C	C3	执行 $(CY) \leftarrow 0$	1	1	×	×	×	√
	CLR bit	C2	执行 $(bit) \leftarrow 0$	2	1	×	×	×	×
	SETB C	D3	执行 $(CY) \leftarrow 1$	1	1	×	×	×	√
	SETB bit	D2	执行 $(bit) \leftarrow 1$	2	1	×	×	×	×
	CPL C	B3	执行 $(CY) \leftarrow (\overline{CY})$	1	1	×	×	×	√
	CPL bit	B2	执行 $(bit) \leftarrow (\overline{bit})$	2	1	×	×	×	×
	ANL C,bit	82	执行 $(CY) \leftarrow (CY) \wedge (bit)$	2	2	×	×	×	√
	ANL C,/bit	B0	执行 $(CY) \leftarrow (CY) \wedge (\overline{bit})$	2	2	×	×	×	√
	ORL C,bit	72	执行 $(CY) \leftarrow (CY) \vee (bit)$	2	2	×	×	×	√
	ORL C,/bit	A0	执行 $(CY) \leftarrow (CY) \vee (\overline{bit})$	2	2	×	×	×	√
	AJMP addr11	Y1	执行 $(PC) \leftarrow (PC) + 2$ $(PC) 10 \sim 0 \leftarrow addr11$	2	2	×	×	×	×

	LJMP addr16	02	执行 $(PC) \leftarrow \text{addr16}$	3	2	×	×	×	×
	SJMP rel	80	执行 $(PC) \leftarrow (PC)+2$ $(PC) \leftarrow (PC)+\text{rel}$	2	2	×	×	×	×
	JMP @A+DPTR	73	执行 $(PC) \leftarrow (A) + (\text{DPTR})$	1	2	×	×	×	×
	JZ rel	60	执行 $(PC) \leftarrow (PC)+2$, 若 $(A)=0$, 则 $(PC) \leftarrow (PC)+\text{rel}$	2	2	×	×	×	×
	JNZ rel	70	执行 $(PC) \leftarrow (PC)+2$, 若 $(A) \neq 0$, 则 $(PC) \leftarrow (PC)+\text{rel}$	2	2	×	×	×	×

续表

指令类型	助记符	十六进制代码	功能	占用字节个数	机器周期数	对标志位的影响			
控制转移类指令	JC rel	40	执行 $(PC) \leftarrow (PC)+2$, 若 $(CY)=1$, 则 $(PC) \leftarrow (PC)+\text{rel}$	2	2	×	×	×	×
	JNC rel	50	执行 $(PC) \leftarrow (PC)+2$, 若 $(CY)=0$, 则 $(PC) \leftarrow (PC)+\text{rel}$	2	2	×	×	×	×
	JB bit,rel	20	执行 $(PC) \leftarrow (PC)+3$, 若 $(\text{bit})=1$, 则 $(PC) \leftarrow (PC)+\text{rel}$	3	2	×	×	×	×
	JNB bit,rel	30	执行 $(PC) \leftarrow (PC)+3$, 若 $(\text{bit})=0$, 则 $(PC) \leftarrow (PC)+\text{rel}$	3	2	×	×	×	×
	JBC bit,rel	10	执行 $(PC) \leftarrow (PC)+3$, 若 $(\text{bit})=1$, 则 $(\text{bit}) \leftarrow 0$ $(PC) \leftarrow (PC)+\text{rel}$	3	2	×	×	×	×
	CJNE A,direct,rel	B5	执行 $(PC) \leftarrow (PC)+3$, 若 $(A) > (\text{direct})$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 0$ $(A) < (\text{direct})$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 1$	3	2	×	×	×	×
	CJNE A,#data,rel	B4	执行 $(PC) \leftarrow (PC)+3$, 若 $(A) > \text{data}$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 0$ $(A) < \text{data}$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 1$	3	2	×	×	×	×
	CJNE Rn,#data,rel	B8~BF	执行 $(PC) \leftarrow (PC)+3$, 若 $(Rn) > \text{data}$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 0$ $(Rn) < \text{data}$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 1$, ($n=0-7$)	3	2	×	×	×	×
	CJNE @Rn,#data,rel	B6,B7	执行 $(PC) \leftarrow (PC)+3$, 若 $((Ri)) > \text{data}$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 0$ $((Ri)) < \text{data}$, 则 $(PC) \leftarrow (PC)+\text{rel}$, $(CY) \leftarrow 1$, ($i=0-1$)	3	2	×	×	×	√
	DJNZ Rn,rel	D8~DF	执行 $(PC) \leftarrow (PC)+2$ 执行 $(Rn) > (Rn)-1$ 若 $(Rn) \neq 0$, 则 $(PC) \leftarrow (PC)+\text{rel}$, ($n=0-7$)	2	2	×	×	×	√
	DJNZ direct,rel	D5	执行 $(PC) \leftarrow (PC)+3$ $(\text{direct}) \leftarrow (\text{direct})-1$ 若 $(\text{direct}) \neq 0$, 则 $(PC) \leftarrow (PC)+\text{rel}$	3	2	×	×	×	×
	ACALL addr11	X1	执行 $(PC) \leftarrow (PC)+2$ $(SP) \leftarrow (SP)+1$ $((SP)) \leftarrow (PCL)$ $(SP) \leftarrow (SP)+1$ $((SP)) \leftarrow (PCH)$ $(PC) 10 \sim 0 \leftarrow \text{addr11}$	3	2	×	×	×	×

续表

指令类型	助 记 符	十六进制代码	功 能	占用字节个数	机器周期数	对标志位的影响			
控制转移类指令	LCALL addr16	12	执行 $(PC) \leftarrow (PC) + 3$ $(SP) \leftarrow (SP) + 1$ $((SP)) \leftarrow (PCL)$ $(SP) \leftarrow (SP) + 1$ $((SP)) \leftarrow (PCH)$ $(PC) \leftarrow \text{addr16}$	3	2	×	×	×	×
	RET	22	执行 $(PCH) \leftarrow (SP)$ $(SP) \leftarrow (SP) - 1$ $(PCL) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$	1	2	×	×	×	×
	RETI	32	执行 $(PCH) \leftarrow (SP)$ $(SP) \leftarrow (SP) - 1$ $(PCL) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$ 从中断返回	1	2	×	×	×	×
	NOP	00	执行空操作, $(PC) \leftarrow (PC) + 1$	1	1	×	×	×	×

16.10 小结

本章详细讲解了 51 系列单片机的指令系统，包括指令的 7 种寻址方式，以及 51 系列单片机指令系统中的各类指令的书写格式、功能、使用方法及注意事项等。对于每一条指令，均给出了完整详细的实例来讲解如何在程序设计中应用。这一章的内容是读者学习使用单片机的基础必备知识，深刻地理解单片机指令系统，可以为接下来的学习打下良好的基础。

第 17 章 51 系列单片机的 定时器/计数器

定时和计数是控制系统中的两个重要功能，也是时序电路的基础。对于时序控制系统，经常需要定时输出某些控制信号，或者对某些待测量进行定时扫描和监测，这便需要实现定时和计数的功能。

51 系列单片机的硬件上集成了可编程的定时器/计数器。对于 MCS-51 子系列单片机，其有两个定时器/计数器，即定时器/计数器 0 和 1，简称 T0 和 T1，有 4 种工作方式可供选择。对于 MCS-52 子系列单片机（如 AT89S52），其有三个定时器/计数器，T0 和 T1 是通用定时器/计数器，定时器/计数器 2（简称 T2）是集定时、计数和捕获三种功能于一体，功能更强。

单片机内部通过专用寄存器 TMOD、TCON 来设置定时器/计数器工作的参数，例如方式选择、定时计数选择、运行控制、溢出标志、触发方式等控制字。本章将详细讲述 51 系列单片机的定时器/计数器的结构、控制寄存器和工作模式。对于每一种工作方式都给出了典型的实例。

17.1 定时器/计数器 0 和 1

定时器/计数器 0 和 1 是 51 系列单片机的两个通用定时器/计数器，简称为 T0 和 T1。T0 和 T1 都具有定时和计数的功能，可以通过特殊功能寄存器来选择。T0 和 T1 共有 4 种工作模式，同样可以通过特殊功能寄存器来选择。下面分别介绍 T0 和 T1 的结构及控制寄存器。

17.1.1 定时器/计数器的结构

定时器/计数器结构的核心是一个 16 位的加 1 计数器，定时器/计数器的结构如图 17-1 所示。其中，这个 16 位的计数器是由两个 8 位计数器组成的，定时器/计数器 T0 由 TH0 和 TL0 构成，T1 由 TH1 和 TL1 构成。TMOD 和 TCON 是定时器/计数器的控制寄存器。

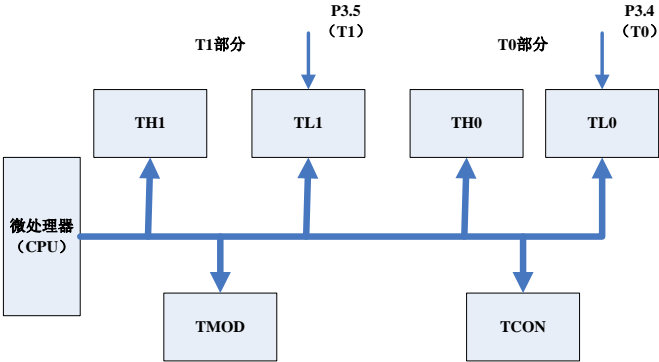


图 17-1 定时器/计数器的结构示意图

寄存器 **TMOD** 主要用于指定各定时器/计数器的功能和工作模式。寄存器 **TCON** 用于控制定时器/计数器的启动和停止计数，同时也设置定时器/计数器的状态。**TMOD** 和 **TCON** 属于特殊功能寄存器，其内容在单片机软件中设置，这样便于单片机程序操作。当单片机进行系统复位时，寄存器 **TMOD** 和 **TCON** 的所有控制位都清零。

对定时器/计数器 **T0** 和 **T1** 每输入一个脉冲，计数器加 1。当 **T0** 或 **T1** 加到计数器每个二进制位全部为 1 时，如果再有脉冲输入，计数器将溢出，此时计数器清零，即全为 0，从而表示定时时间已到或计数值已满。此时，在寄存器 **TCON** 中置位中断请求标志位 **TF0** 或 **TF1**，向 CPU 发出中断处理申请，用户便可以在程序中进行相应的代码处理。

17.1.2 定时器/计数器的功能

对于 51 系列单片机，定时器/计数器 **T0** 和 **T1** 都具有定时和计数两种功能，可以在定时器模式控制寄存器 **TMOD** 中的控制位 C/\bar{T} 中进行设置。下面分别介绍这两种功能的工作原理。

1. 计数

计数就是对计数脉冲进行计数。其中，计数脉冲来自相应的外部输入引脚 **P3.4** (**T0**) 或 **P3.5** (**T1**)。当该引脚的输入信号发生由高电平至低电平的负跳变时，计数器 (**TH0**、**TL0** 或 **TH1**、**TL1**) 的值增加 1。

单片机对外部输入信号的占空比没有要求，但一般要使这一电平至少保持一个机器周期，主要是为了确保所给定的电平在变化前至少被采样一次。单片机 CPU 在每个机器周期的 S_5P_2 期间，对外部脉冲信号输入进行采样。如果在第一个周期的采样值为 1，而在下一个周期中的采样值为 0，即检测到引脚电平的负跳变，则在下一个周期的 S_3P_1 期间，计数值增加 1。由于总共需要两个机器周期来确认一次计数，即需要 24 个振荡器周期，因此外部输入的计数脉冲的最高频率为单片机振荡器频率的 $1/24$ 。

2. 定时

定时是对时间进行统计。定时器/计数器的定时功能其实也是通过计数实现的，与计数功能不同的是，此时计数脉冲来自于单片机的内部时钟脉冲。

单片机在每个机器周期都将计数器的值增加 1，由于每个机器周期等于 12 个振荡器周期，因此计数器的计数速率为振荡器频率的 $1/12$ 。当晶体振荡频率选定时，机器周期也就确定了。例如，当采用振荡频率为 24MHz 晶振时，机器周期 $0.5\mu s$ ，计数速率为 2MHz，即每隔 $0.5\mu s$ 计数器加 1，这样就可以将对机器周期的计数转换成对时间的计数。计数值乘以单片机的机器周期即为定时时间。

17.1.3 T0 和 T1 的控制寄存器

定时器/计数器 **T0** 和 **T1** 用到的控制和状态寄存器为 **TMOD** 和 **TCON**，其主要用于定义定时器/计数器的功能和工作模式。单片机 CPU 每执行一条涉及 **TMOD** 和 **TCON** 的指令时，例如工作方式选择、定时计数模式选择、溢出标志、触发方式等，该操作将在下一条指令的第一个机器周期 S_1P_1 开始执行。

1. 定时器/计数器 0 和 1 的方式控制寄存器 TMOD

定时器/计数器 0 和 1 的方式控制寄存器 **TMOD**，如图 17-2 所示。

TMOD 寄存器用于定义 **T0** 和 **T1** 的工作方式和 4 种工作模式。其内容分为两部分，低 4 位用于控制 **T0**，高 4 位用于控制 **T1**，两部分操作和含义完全相同。下面分别介绍各种控制位

的含义。

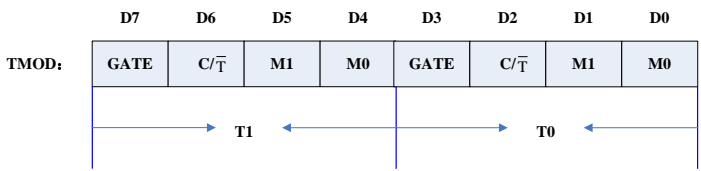


图 17-2 控制寄存器 TMOD

➤ 门控制位 GATE

门控制位 GATE=1 时，定时器/计数器的运行受外部引脚输入电平的控制。其中 $\overline{INT0}$ 引脚控制 T0， $\overline{INT1}$ 引脚控制 T1。当控制引脚为高电平且 TR0 或 TR1 置 1 时，相应的定时器/计数器才被选通。

门控制位 GATE=0 时，只要 TR0 或 TR1 置 1，相应的定时器/计数器就被选通，此时不受外部输入引脚的控制。

➤ 工作方式选择位 C/ \overline{T}

工作方式选择位 C/ \overline{T} =0 时，为定时器工作方式。单片机 CPU 采用晶振脉冲的 12 分频信号作为计数器的计数信号，即对机器周期进行计数。如果外接 12MHz 晶振，则该定时器的计数频率为 1MHz。

工作方式选择位 C/ \overline{T} =1 时，为计数器工作方式。此时单片机的计数脉冲为外部引脚 P3.4 (T0) 或 P3.5 (T1) 的输入脉冲。当 T0 或 T1 输入发生从高到低的负跳变时，相应的计数器加 1。在 51 系列单片机中，最高计数频率为晶振频率的 1/24。

➤ 工作模式选择位 M1、M0

工作模式选择位 M1 和 M0 用于设置定时器/计数器的工作模式。M1 和 M0 共两位，对应 4 种工作模式。定时器/计数器的工作模式，如表 17-1 所示。

表 17-1 定时器/计数器的工作模式

M1	M0	工作模式	功能说明
0	0	模式 0	13 位定时器/计数器
0	1	模式 1	16 位定时器/计数器
1	0	模式 2	自动重新装入的 8 位定时器/计数器
1	1	模式 3	T0 分成两个 8 位计数器，T1 停止计数

TMOD 寄存器的单元地址为 89H，不能位寻址，只能用字节传送指令设置其内容。复位后 TMOD 的所有位均为 0，即自动设置为定时器方式和工作模式 0。在汇编语言程序中，可以采用如下语句设置 TMOD 寄存器。

```
MOV      TMOD, #10H           ;选择 T1，定时模式 1
```

该语句设置 TMOD=0x10=00010000B，即选择定时器/计数器 T1，其工作于模式 1，即 16 位定时器/计数器模式。在 C51 语言中可以直接为其赋值。示例如下：

```
sfr TMOD = 0x89;              //声明特殊功能寄存器 TMOD
TMOD=0x20;                    //初始化 T1 为定时功能，模式 2
```

该语句设置 TMOD=0x20=00100000B，即选择定时器/计数器 T1，其工作于模式 2，即自动重新装入的 8 位定时器/计数器。其中，对 TMOD 的声明，一般包含在头文件 REG51.H 中。

2. 定时器/计数器 0 和 1 的中断控制寄存器 TCON

定时器/计数器 0 和 1 的中断控制寄存器 TCON，如图 17-3 所示。

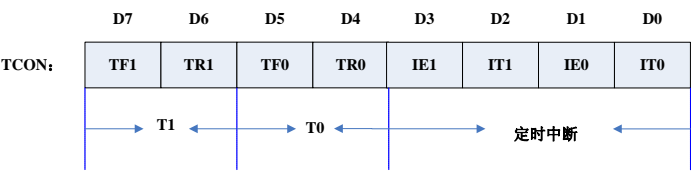


图 17-3 控制寄存器 TCON

寄存器 TCON 的功能是在定时器溢出时设定标志位，并控制定时器的运行、停止和中断请求。其包含三个部分，TF1 和 TR1 位用于控制 T1，TF0 和 TR0 位用于控制 T0，其他的为中断控制。下面分别介绍各种控制位的含义。

➤ 溢出标志位 TF1/TF0

当定时器 T1/T0 溢出时，硬件自动将 TF1/TF0 置 1，并申请中断。当进入中断服务程序时，硬件又将自动清零 TF1/TF0。

➤ 启/停控制位 TR1/TR0

该位由软件置位和复位。当 GATE 为 0 时，TR1/TR0 置位为 1 时 T1/T0 开始计数，TR1/T0 复位为 0 时 T1/T0 停止计数；当 GATE 为 1 时，TR1/TR0 为 1 且 $\overline{\text{INT1}}/\overline{\text{INT0}}$ 输入高电平时，T1/T0 开始计数。

寄存器 TCON 的低 4 位与外部中断有关，将在下一章介绍。

TCON 的字节地址 88H，可以位寻址，位地址为 88H~8FH。因此，在汇编语言程序中，可以采用如下语句设置 TCON 寄存器。

```
SETB TR0                                ;启动定时器/计数器 T0
```

该语句采用位寻址的方式，置位 TR0，用于启动定时器/计数器 T0。

在 C51 语言中，头文件 REG51.H 包含了 TCON 的位定义，示例如下：

```
sbit      TF1   = 0x8F;                    //溢出标志位 TF1
sbit      TR1   = 0x8E;                    //启/停控制位 TR1
sbit      TF0   = 0x8D;                    //溢出标志位 TF0
sbit      TR0   = 0x8C;                    //启/停控制位 TR0
sbit      IE1   = 0x8B;                    //外部中断 1 请求标志位
sbit      IT1   = 0x8A;                    //外部中断 1 的中断触发方式控制位
sbit      IE0   = 0x89;                    //外部中断 0 请求标志位
sbit      IT0   = 0x88;                    //外部中断 0 的中断触发方式控制位
```

因此，在 C51 语言可以直接对这些位进行操作，示例如下：

```
TR0=1                                    //启动定时器/计数器 T0
```

该语句直接置位 TR0，用于启动定时器/计数器 T0。

3. 定时器/计数器的初始化

51 系列单片机的定时器/计数器是编程控制的，这对于定时和计数的使用非常方便。在使用之前要先进行初始化，步骤如下：

- (1) 赋值 TMOD 寄存器，用来指定定时/计数器的工作模式；
- (2) 赋值 TH0、TL0 或 TH1、TL1，用来装入定时/计数器的初值；

- (3) 如果需要使用中断，则可以对 IE 赋值，启动定时器中断；
 - (4) 对 TR0、TR1 置位，启动定时/计数器，置位后，计数器即按规定的工作模式和初值进行计数或开始定时。
- 如果采用汇编语言进行程序设计，按照前面的步骤，可以用 MOV 指令和 SETB 指令来初始化定时器/计数器。程序示例如下：

```
MOV      TMOD, #06H           ;定时器 T0 工作于方式 2，外部计数
MOV      TH0, #0FFH          ;初值
MOV      TL0, #0ABH
SETB TR0                      ;启动 T0
```

如果采用 C51 语言进行程序设计，则可以按照前面的步骤，直接采用赋值语句来初始化定时器/计数器。程序示例如下：

```
TMOD=0x06;                    //初始化 T0 工作于方式 2，外部计数
TL0=0x0AB;                     //设初值
TH0=0x0FF;
TR0=1;                         //启动定时器
```

由于计数器是加法计数，其在计数溢出时申请中断，因此不能直接输入所需计数值，而是应从计数最大值计算得出需要置入的初值。假设计数最大值为 MAX（在不同的工作模式中，MAX 可以为 2^{13} 、 2^{16} 和 2^8 ），则置入的初值 X 计算方法如下：

- 定时方式时，满足等式 $(MAX-X) \times T = \text{定时值}$ ，因此 $X = MAX - \text{定时值} / T$ 。
- 计数方式时， $X = MAX - \text{计数值}$ 。

式中 T 为计数周期，即单片机的机器周期。例如当 $T = 0.5\mu s$ 时，在工作模式 0 时， $MAX = 2^{13} \times 0.5\mu s = 4.096ms$ ；在工作模式 1 时，则最大定时值为 $MAX = 2^{16} \times 0.5\mu s = 32.768ms$ 。

17.2 定时器/计数器 0 和 1 的工作模式

51 系列单片机的定时器/计数器具有丰富的功能，T0 和 T1 除了可以选择计数或定时工作方式外，每个定时器/计数器还有 4 种工作模式（模式 0、模式 1、模式 2 和模式 3）。通过对寄存器 TMOD 的 M1 和 M0 位进行编码，选择 4 种工作模式中的任一种。

定时器/计数器 T0 和 T1 在模式 0~2 工作时，用法是一样的，仅在模式 3 时有所区别。下面分别介绍这 4 种工作模式。

17.2.1 工作模式 0 及其程序设计

工作模式 0 是 13 位计数器，其由一个高 8 位（0~7）计数器（TH0 或 TH1）和一个具有 32 位分频的低 8 位计数器中的（TL0 或 TL1）的低 5 位（0~4）组合成。

1. 工作模式 0

在工作模式 0 中，T1 和 T0 的组成结构与功能完全相同。这里以 T0 为例，T0 的组成结构如图 17-4 所示。当置寄存器 TMOD 的 M1=0、M0=0 时，T0 和 T1 就工作在模式 0 状态。

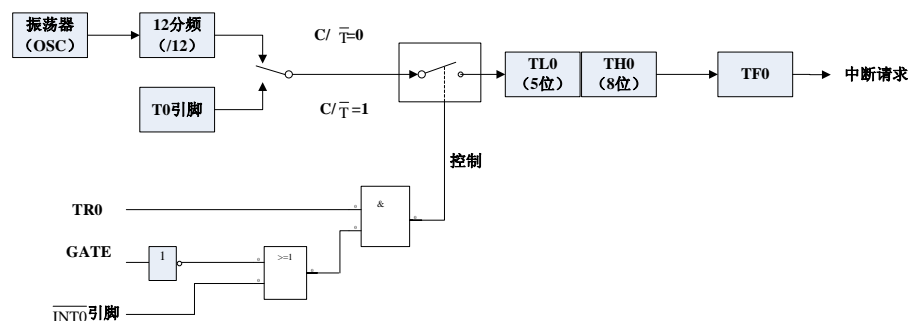


图 17-4 T0 的工作模式 0 示意图

在工作模式 0 下，无论工作于计数方式还是定时方式，其运行原理都是一样的。当 TL0 (或 TL1) 的低 5 位计数溢出时，向 TH0 (或 TH1) 进位，当 TH0 (或 TH1) 计数器溢出时，计数器清零，并向溢出控制位 TF0 (或 TF1) 进位置 1，同时向 CPU 发出中断请求。

由图 17-4 可见，在模式 0 状态下，门控制位 GATE 和工作方式选择位 C/\bar{T} 决定了其工作状态，其原理如下：

- 当 $C/\bar{T}=0$ 时，多路开关接通内部振荡器的 12 分频输出，此时计数时钟由晶体振荡器 12 分频产生，13 位计数器就是对机器周期进行计数，即为定时器工作方式，其定时时间 = $(2^{13} - T0 \text{ 初值}) \times \text{时钟周期} \times 12$ 。
- 当 $C/\bar{T}=1$ 时，多路开关接通计数引脚 T0 (或 T1)，外部计数脉冲由 P3.4 (T0) 或 P3.5 (T1) 输入。当计数脉冲发生负跳变时，计数器加 1，即为计数器工作方式。
- 当 GATE=0 时，经反相后使或门输出 1，此时仅由 TR0 (或 TR1) 控制与门的开启；当 TR0 (或 TR1)=1 时，与门输出 1，控制开关闭合，启动计数器工作，直至溢出，溢出时计数器清零，TF0 (或 TF1)=1，向 CPU 申请中断；当 TR0 (或 TR1)=0 时，控制开关断开，计数器停止工作。一般情况常使用这种方式。
- 当 GATE=1 时，则由 $\overline{INT1}$ (或 $\overline{INT0}$) 控制或门的输出，此时与门的开启由 $\overline{INT1}$ (或 $\overline{INT0}$) 和 TR0 (或 TR1) 共同控制，当 TR0 (或 TR1)=1 时，外部中断信号通过 $\overline{INT1}$ (或 $\overline{INT0}$) 直接控制定时/计数器的启动和停止，当 $\overline{INT1}$ (或 $\overline{INT0}$) 由 0 变为 1 时，启动计数；当 $\overline{INT1}$ (或 $\overline{INT0}$) 由 1 变为 0 时，停止计数。当 $\overline{INT1}$ (或 $\overline{INT0}$)=1 启动计数， $\overline{INT1}$ (或 $\overline{INT0}$)=0 停止计数，这样就记录了一个脉冲的宽度，这种方法常用来测量在 $\overline{INT1}$ (或 $\overline{INT0}$) 端出现的正脉冲的宽度。

2. 定时器/计数器模式 0 的应用及程序设计

下面举例讲解定时器/计数器模式 0 在具体程序设计中的应用。假定单片机外接 6MHz 晶振，这里采用定时器 T0，采用模式 0 使定时器产生 1ms 的定时，并在 P1.1 端口输出周期为 2ms 的方波。

首先，计算单片机的机器周期为 $12 \div 6\text{MHz} = 2 \times 10^{-6}\text{s} = 2\mu\text{s}$ 。然后，计算定时器初值 X，则 $(2^{13} - X) \times 2 \times 10^{-6} = 1 \times 10^{-3}$ 。从而，求得 $X = 7692 = 1\text{E}0\text{CH}$ 。

根据 13 位定时器的结构特性，定时器 T0 的初值应该设置为 TH0=0F0H，TL0=0CH。如果采用汇编语句进行程序设计，其程序示例如下：

```
ORG      0000H
AJMP START                                ;转向主程序
ORG      000BH
AJMP ITRU                                ;转向中断子程序
ORG      0200H
```

51 单片机开发与应用技术详解

```
START:  MOV      SP,#60H
        MOV      TMOD,#00H           ;设置定时器 0 模式 0
        MOV      TL0,#0CH           ;初始化
        MOV      TH0,#0F0H
        SETB TR0
        SETB ET0
        SETB EA                       ;开中断
HERE:   SJMP HERE
ORG     0400H
ITRU:   MOV      TL0,#0CH           ;重置初值
        MOV      TH0,#0F0H
        CPL      P1.1              ; P1.1 输出取反
        RETI
        END
```

在上面的程序中采用了中断服务程序产生要求的方波，在中断服务子程序中，首先将定时器初值再次装入定时器，以便于开始下一次定时，然后通过取反 P1.1 得到方波波形。如果采用 C51 语言进行程序设计，其程序示例如下：

```
#include <reg51.h>                               //头文件

sbit b=P1^1;                                       //位定义

void T0ISR(void) interrupt 1                       //定时器 T0 中断响应
{
    TL0=0x0C;                                     //重置计数初值
    TH0=0x0F0;
    b=~b;                                         //反向
}

void main(void)                                    //主函数
{
    b=0;                                         //初始化 P1^1=0
    TMOD=0x00;                                  //设置定时器 T0 为模式 0
    TL0=0x0C;                                   //初始化
    TH0=0x0F0;
    TR0=1;
    ET0=1;
    EA=1;                                       //开中断
    while(1)                                    //主循环
    {
    }
}
```

在上面的程序中，将位变量 b 指向 P1^1。在主程序中初始化定时器 T0，接着开相应的中断，并进入主循环。主循环中不进行任何操作。定时器溢出时将触发中断，在中断函数 T0ISR 中首先重置计数初值，接着将 P1^1 反向。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

17.2.2 工作模式 1 及其程序设计

工作模式 1 与模式 0 的区别在于计数的位数不同，模式 1 是由 TH0(或 TH1)的高 8 位(0~7)和 TL0(或 TL1)的低 8 位(0~7)组合成的 16 位计数器。

1. 工作模式 1

当置 M1=0 且 M0=1 时，定时器/计数器工作于模式 1 状态，此时 T0 的组成结构，示意图

如图 17-5 所示。在同一模式中，T1 和 T0 的组成结构与功能完全相同。

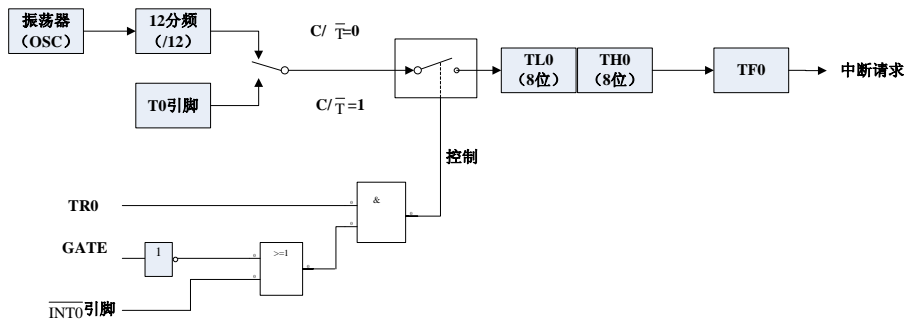


图 17-5 T0 的工作模式 1 示意图

由图 17-5 中看出，门控制位 GATE 和工作方式选择位 C/\overline{T} 决定了其工作状态。这些控制位的含义和模式 0 相同，可以参阅模式 0 的介绍。模式 1 的定时时间 = $(2^{16} - T0 \text{ 初值}) \times \text{时钟周期} \times 12$ 。其计数范围更大，因此实际中多数采用工作模式 1。

2. 定时器/计数器模式 1 的应用及程序设计

定时器/计数器模式 1 和模式的工作原理相同，只不过模式 1 使用 16 位的计数器，可以具有较长的定时周期。下面举例讲解定时器/计数器模式 1 在具体程序设计中的应用。

假定单片机外接 12MHz 的晶振，选用定时器/计数器 1，每隔 10ms 将 P1.1 端口反向，即在端口 P1.1 输出周期为 20ms 的方波。由于定时时间为 10ms，可以求得定时器 1 的低 8 位初值为 0F0H，高 8 位初值为 0D8H。

如果采用汇编语言进行程序设计，其程序示例如下：

```
ORG      0000H
AJMP MAIN                                ;转向主程序段
ORG      001BH
AJMP T1INT                                ;转向中断程序
ORG      0200H
MAIN:    MOV     SP,#60H                    ;设置堆栈区域
          MOV     TMOD,#10H                 ;选择 T1，定时模式 1
          MOV     TH1,#0D8H                 ;初始化
          MOV     TL1,#0F0H
          SETB EA                            ;开中断
          SETB ET1
          SETB TR1                            ;启动 T1
HERE:    SJMP HERE
T1INT:    MOV     TL1,#0F0H                 ;重置初值
          MOV     TH1,#0D8H
          CPL     P1.1                       ; P1.1 输出取反
          RETI
END
```

上面的程序中首先初始化定时器/计数器 T1，然后开中断。在中断服务子程序中，将定时器初值再次装入定时器 T1，以便于开始下一次定时，然后反向 P1.1 端口。如果采用 C51 语言进行程序设计，其程序示例如下：

```
#include <reg51.h>                        //头文件

sbit b=P1^1;                               //位定义
```

```
void T1ISR(void) interrupt 3 //定时器 T1 中断响应
{
    TL1=0x0F0; //重置计数初值
    TH1=0x0D8;
    b=~b; //反向
}

void main(void) //主函数
{
    b=0; //初始化 P1^1=0
    TMOD=0x10; //设置定时器 T1 为模式 1
    TL1=0x0F0; //初始化
    TH1=0x0D8;
    TR1=1;
    ET1=1;
    EA=1; //开中断
    while(1) //主循环
    {
    }
}
```

在上面的程序中，将位变量 **b** 指向 **P1^1**。在主程序中初始化定时器 **T1**，接着开相应的中断，并进入主循环。主循环中不进行任何操作。定时器溢出时将触发中断，在中断函数 **T1ISR** 中首先重置计数初值，接着将 **P1^1** 反向。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

17.2.3 工作模式 2 及其程序设计

工作模式 2 是将 **TL0**（或 **TL1**）配制成自动重新装入的 8 位计数器。

1. 工作模式 2

当置 **M1=1** 且 **M0=0** 时，定时器/计数器工作于模式 2 状态。在工作模式 2 时，**T0** 的组成结构示意图如图 17-6 所示。在同一模式中，**T1** 和 **T0** 的组成结构与功能完全相同。

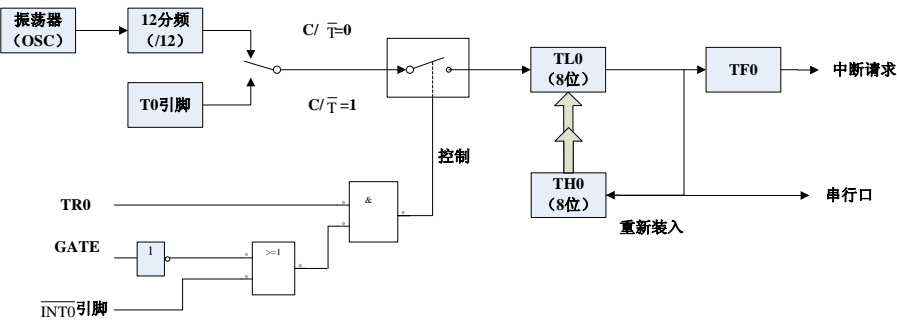


图 17-6 T0 的工作模式 2 示意图

由图 17-6 可见，模式 3 中，16 位计数器分成独立的两个部分，其中 **TH0**（或 **TH1**）作为计数初值寄存器，用于存放和保持初值，初值在程序中设置。

装入初值并启动定时器/计数器后，**TL0**（或 **TL1**）按加法计数器工作，当 **TL0**（或 **TL1**）溢出时，使溢出标志位 **TF0**（或 **TF1**）进位置 1，并选通三态门，将 **TH0**（**TH1**）存放的初值

重新装入到 TL0（或 TL1）中，使 TL0（或 TL1）从初值开始继续下一轮的计数，如此循环下去。作为额外的用途，溢出信号还将送到串行通信系统，设置并产生串行通信波特率。

模式 2 具有使 8 位计数器 TL0（或 TL1）自动重装初值的功能，在程序初始化时，为 TH0（TH1）、TL0（或 TL1）赋同样的初值，此后 TL0（或 TL1）每次溢出都由 TH0（TH1）重新装入初值。在这个过程中，不会丢失计数信号，且重新再装入也不会影响 TH0（TH1）中存放的初值。

由于模式 2 的自动重装初始值的特性，使其对定时控制很有用，特别适用于计数范围较小、较精确的脉冲信号发生器，常作为串行口波特率发生器使用。其周期=（ 2^8 -T0 初值）×时钟周期×12。

2. 定时器/计数器模式 2 的应用及程序设计

定时器/计数器模式 2 是自动重装模式，在这种模式下，计数器初值只需设置 1 次，在定时器溢出后不用软件重新设置，硬件会自动复位。下面举例讲解定时器/计数器模式 2 在具体程序设计中的应用。

假设需要产生如图 17-7 所示的波形。P2.0 引脚和 P2.1 引脚初始值分别为高电平和低电平。其中一个低频脉冲信号从引脚 P3.4（T0）输入单片机，该信号的频率小于 1kHz，即周期大于 1ms。当 P3.4 引脚电平发生负跳变的时候，触发 P2.0 引脚输出一个 500μs 的低电平信号，同时触发 P2.1 引脚输出一个 1000μs 的高电平信号。最后，P2.0 引脚和 P2.1 引脚恢复初始状态。

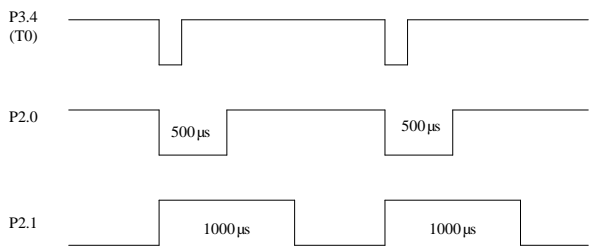


图 17-7 波形图

这里采用汇编语言进行程序设计，程序示例如下：

```

                                ORG      0000H
MAIN:  MOV      TMOD, #06H      ;定时器 T0 工作于方式 2，外部计数
                                MOV      TH0, #0FFH      ;初值，计数为 01 就溢出
                                MOV      TL0, #0FFH
                                CLR       P2.1          ;置 P2.1 为 0
                                SETB TR0                ;启动 T0
DEL1:  JBC      TF0, RRZ1        ;检查外部信号
                                AJMP DEL1
RRZ1:  CLR      TR0
                                MOV      TMOD, #02H      ;重置 T0 为 500μs 定时
                                MOV      TH0, 0AH
                                MOV      TL0, 0AH
                                SETB P2.1
                                CLR       P2.0
                                SETB TR0                ;启动定时器
DEL2:  JBC      TF0, RRZ2        ;首次检测 500μs
                                AJMP DEL2
RRZ2:  SETB P2.0
DEL3:  JBC      TF0, RRZ3        ;第二次检测 500μs
```

```

    AJMP DEL3
RRZ3:  CLR      P2.1
        CLR      TR0
        AJMP MAIN           ; 返回主函数
        END
```

在上面的程序中，首先在主程序部分设置定时器 T0 工作于模式 2，采用外部计数，并设置初值，使其计数为 01 就溢出。然后接着重置 T0 为 500μs 定时，分别检测第一次和第二次溢出，并相应地为 P2.0 和 P2.1 赋值，从而实现需要的波形输出。

另外，工作模式 2 常作为串行口波特率发生器使用。下面采用 C51 语言，来讲解如何使用工作于模式 2 的定时器/计数器 T1 作为串行通信的波特率发生器。程序示例如下：

```

#include<ctype.h>           //头文件
#include<stdio.h>
#include <reg51.h>

void main(void)             //主函数
{
    unsigned char i;        //变量声明
    char *ch;

    SCON=0x50;              //串口模式 1，允许接收
    TMOD|=0x20;             //初始化 T1 为定时功能，工作于模式 2
    PCON|=0x80;             //设置 SMOD=1
    TL1=0xF4;               //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;               //中断
    TR1=1;                  //启动定时器

    for(i=0;i<128;i++)      //循环输出
    {
        ch=(isalpha(i)? "YES":"NO"); //调用 isalpha 函数
        printf("isalpha(%c)=%s\n",i,ch); //输出结果
    }
}
```

在上面的程序中，初始化 T1 为定时功能，工作于模式 2，并设置 TL1 和 TH1 的初值来产生特定的波特率。由于工作模式 2 在溢出时可以自动重装。因此，只要设置初值后，便可以一直工作。程序后面采用串口输出字符串。有关串口的相关内容，可以参阅后面章节的介绍。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

17.2.4 工作模式 3 及其程序设计

工作模式 3 是将 16 位计数器分成两个相互独立的 8 位计数器 TL0 和 TH0。定时器/计数器的工作模式 3 只适用于 T0。对于 T1，设置为模式 3 时，相当于使 TR1=0，使其停止计数，没有什么实际意义。

1. 工作模式 3

当 TMOD 的低两位被置位为 1 时，T0 被设置为模式 3。当 T0 处于模式 3 时，T0 的内部组成结构示意图如图 17-8 所示。

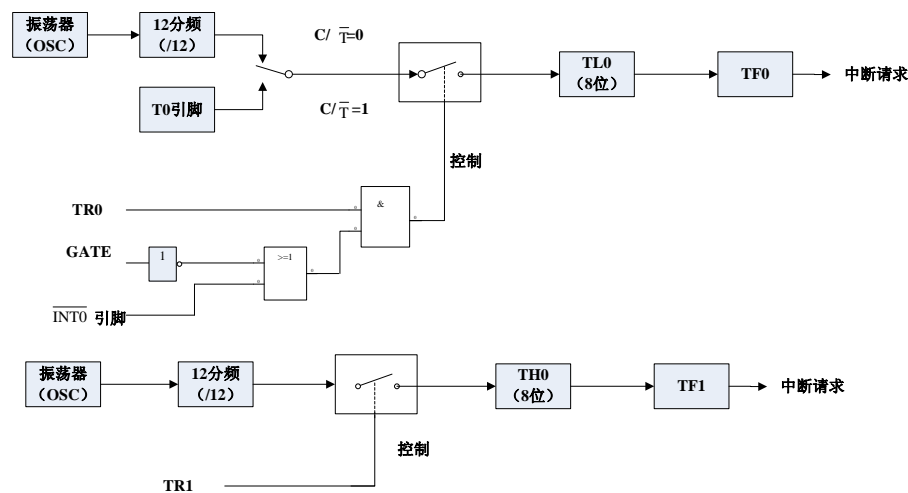


图 17-8 T0 的工作模式 3 示意图

由图 17-8 可见，TL0 使用了 T0 的状态控制位 C/\overline{T} 、GATE、TR0、 $\overline{INT0}$ 和 TF0。其操作情况与模式 0 和模式 1 相同，即可以按计数方式工作，也可以按定时方式工作。

在工作模式 3 下，TH0 被固定为一个只能按定时方式工作的 8 位定时器，即只对机器周期计数，并使用了 T1 的状态控制位 TR1 和 TF1，并占用了 T1 的中断。因此 TH0 的启动和停止受 TR1 控制，TH0 的溢出将置位 TF1。

模式 3 适用于计数范围较小且要求增加一个附加的 8 位定时器的情况，使单片机具有三个定时器/计数器。例如有些场合需要两个计数范围在 256 以内的定时/计数，同时又需要提供串行通信的波特率产生。此时可将 T0 设置为模式 3，TH0 控制了 T1 的中断，此时 T1 可以设置为模式 0、模式 1 或模式 2，用在作为串行口的波特率发生器或不需要中断控制的场合。

2. 定时器/计数器模式 3 的应用及程序设计

对于定时器/计数器模式 3，THx 和 TLx 分别作为两个独立的 8 位定时器使用。下面举例讲解该模式在具体程序设计中的应用。

假定单片机需要产生如图 17-9 所示的波形。单片机外接 6MHz 晶振，启用定时器 T0，工作于模式 3。程序中分别采用 TL0 和 TH0 产生 200μs 和 400μs 的定时中断，从而使 P1.0 引脚和 P1.1 引脚输出 400μs 和 800μs 的方波。

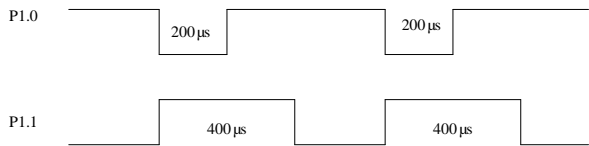


图 17-9 波形图

由于晶振频率为 6MHz，因此单片机的机器周期为 2μs。计算两个计数器的初值 X1 和 X2。X1 和 X2 满足如下等式。

$$200 = (2^8 - X1) \times 2$$

$$400 = (2^8 - X2) \times 2$$

从而求得 X1=9CH，X2=38H。

如果采用汇编语言进行程序设计，其程序示例如下：

51 单片机开发与应用技术详解

```

    ORG      0000H
MAIN:  AJMP START      ;转主程序
    ORG      000BH
    AJMP RIT0          ;T0 中断入口
    ORG      001BH
    AJMP RIT1          ;转向 T0 中断服务程序
    ORG      0100H
START: MOV      SP,#60H
    ACALL DDZ
HERE:  SJMP HERE
DDZ:   MOV      TMOD,#03H      ;T0 初始化
    MOV      TL0,#9CH        ;赋初值
    MOV      TH0,#38H        ;赋初值
    SETB TR0                 ;启动定时器/计数器 T0
    SETB ET0                 ;允许 T0 中断
    SETB TR1                 ;启动定时器/计数器 T1
    SETB ET1                 ;允许 T1 中断
    SETB EA                  ;开启中断
    RET
RIT0:  MOV      TL0,#9CH      ;T0 中断服务例程
    CPL      P1.0
    RETI
RIT1:  MOV      TH0,#38H      ;T1 中断服务例程
    CPL      P1.1
    RETI
END
```

在上面的程序中，设置 T0 工作于模式 3，而 T1 工作与模式 0。主程序中对 T0 和 T1 进行初始化，而在 T0 和 T1 的中断服务例程中分别重置初值，并为 P1.0 和 P1.1 赋值。如果采用 C51 语言，其程序代码示例如下：

```

#include <reg51.h>                //头文件

sbit a=P1^0;                      //定义位变量
sbit b=P1^1;

void T0ISR(void) interrupt 1      //定时器 T0 中断响应
{
    TL0=0x9C;                    //重置计数初值
    a=~a;
}

void T1ISR(void) interrupt 3      //定时器 T1 中断响应
{
    TH0=0x38;                   //重置计数初值
    b=~b;
}

void main(void)                  //主函数
{
    b=0;                        //初始化 P1^1=0
    TMOD=0x03;                  //设置定时器 T0 为模式 3
    TL0=0x9C;                   //初始化
```

```
TH0=0x38;
TR0=1;
ET0=1;
TR1=1;
ET1=1;
EA=1;                                //开中断
while(1)                              //主循环
{
}
}
```

在程序中，首先设置 T0 工作于模式 3，T1 工作与模式 0，然后开中断，接着进入主循环。在主循环中不做任何事情，当 T0 和 T1 中断发生时，分别在其中断服务例程中重置初值并为端口赋值。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

17.3 定时器/计数器 2

定时器/计数器 2 是一个 16 位的具有自动重载和捕获能力的定时器/计数器，在 52 系列单片机中提供。52 子系列单片机是对 51 子系列的扩展，在其特殊功能寄存器组中与 T2 有关的寄存器分别为控制寄存器 T2CON 和 T2MOD、捕获寄存器 RCAP2H 和 RCAP2L、定时寄存器高低字节 TH2 和 TL2。

17.3.1 T2 的控制寄存器 T2CON 和 T2MOD 及其程序访问

T2 的特殊功能寄存器 T2CON 和 T2MOD，同定时器/计数器 T0 和 T1 的 TCON 和 TMOD 类似。下面分别进行介绍。

1. 控制寄存器 T2CON

控制寄存器 T2CON 用于设定 T2 的工作方式、各种功能选择及有关状态信息。T2CON 的格式及各状态位定义，如图 17-10 所示。

	D7	D6	D5	D4	D3	D2	D1	D0
T2CON:	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T $\overline{2}$	CP/RL $\overline{2}$

图 17-10 T2CON 的格式定义

下面分别介绍 T2CON 各位的含义。

- CP/RL $\overline{2}$ (D0): 定时器/计数器 2 的捕获或重新再装入选择位。当 CP/RL $\overline{2}$ =1 时，如果 EXEN2=1，则在 T2EX (P1.1) 引脚上的负跳变将触发捕获操作，即将 TH2 和 TL2 的内容传递给 RCAP2H 和 RCAP2L；当 CP/RL $\overline{2}$ =0 时，若 EXEN2=1，则 T2 计数满回 0 溢出或 T2EX (P1.1) 引脚上的负跳变，都将触发重新再装入操作，即将 RCAP2H 和 RCAP2L 的内容传递给 TH2 和 TL2；当 RCLK=1 或 TCLK=1 时，CP/RL $\overline{2}$ 标志位不起作用。T2 的计数满回 0 溢出时，将强制 T2 进行自动重新再装入操作。重新再装入操作常用于串行口的波特率发生器。CP/RL $\overline{2}$ 位须由软件置位或复位。
- C/T $\overline{2}$ (D1): 定时器/计数器 2 的定时或计数功能选择位。当 C/T $\overline{2}$ =1 时，选择计数器工作方式，下降沿触发，计数脉冲来自于外引脚 T2CLK；当 C/T $\overline{2}$ =0 时，选择为定时器工作方式，做波特率发生器时，对 f_{osc}/2 计数，不做波特率发生器时，对 f_{osc}/12 计数。C/T $\overline{2}$ 位须由软件置位或复位。
- TR2 (D2): 定时器/计数器 2 的启动停止控制位。当软件置位 TR2 为 1 时，启动 T2 开

始计数；复位 TR2 为 0 时，停止计数。

- EXEN2 (D3)：定时器/计数器 2 的外部触发允许标志位。当 EXEN2=1 时，如果 T2 不是正工作在串行通信端口的时钟，则在 T2EX 引脚（P1.1）上的负跳变将触发捕获或重新再装入，并置 EXF2 位为 1，请求中断；当 EXEN2=0 时，在 T2EX 引脚（P1.1）上的负跳变对 T2 不起作用。
- TCLK (D4)：串行口发送时钟标志位。当 TCLK=1 时，串行通信端使用 T2 的回 0 溢出信号作为串行口 1 和 3 的发送时钟；当 TCLK=0 时，使用 T1 的回 0 溢出信号作为发送时钟。TCLK 位须由软件置位或复位。
- RCLK (D5)：串行口接收时钟标志位。当 RCLK=1 时，串行通信端使用 T2 的回 0 溢出脉冲作为串行口 1 和 3 的接收时钟；当 RCLK=0 时，使用 T1 的回 0 溢出脉冲作为接收时钟。RCLK 位须由软件置位或复位。
- EXF2 (D6)：定时器/计数器 2 回外部中断请求标志位。当 EXEN2=1 且 T2EX 引脚上出现负跳变而引起捕获或重新再装入时，则 EXF2 置位，并向 CPU 申请中断，此时若允许 T2 中断，CPU 将响应中断，并转向中断服务程序，EXF2 也必须由软件复位。
- TF2 (D7)：定时器/计数器 2 的溢出中断请求标志位。当 T2 计数溢出复位为 0 时，由内部硬件置位 TF2，并申请中断。但在波特率发生器方式下，当 RLCK 位或 TCLK 置位为 1 时，T2 计数溢出将不对 TF2 置位，此时必须由软件复位。

T2 的字节地址为 C8H，可以位寻址，所有标志位都可以用操作指令来置位或清零，其位地址为 C8H~CFH。因此，在汇编语言程序中，可以采用如下语句设置 T2CON 寄存器。

```
SETB    TR2                                ;启动定时器/计数器 T2
```

该语句采用位寻址的方式，置位 TR2，用于启动定时器/计数器 T2。

在 C51 语言中，头文件 REG52.H 包含了 T2CON 的位定义，示例如下：

```
sfr      T2CON    = 0xC8;                //定义 T2CON
sbit     TF2      = T2CON^7;              //定时器/计数器 2 的溢出中断请求标志位
sbit     EXF2     = T2CON^6;              //定时器/计数器 2 回外部中断请求标志位
sbit     RCLK     = T2CON^5;              //串行口接收时钟标志位
sbit     TCLK     = T2CON^4;              //串行口发送时钟标志位
sbit     EXEN2    = T2CON^3;              //定时器/计数器 2 的外部触发允许标志位
sbit     TR2      = T2CON^2;              //定时器/计数器 2 的启动停止控制位
sbit     C_T2     = T2CON^1;              //定时器/计数器 2 的定时或计数功能选择位
sbit     CP_RL2   = T2CON^0;              //定时器/计数器 2 的捕获或重新再装入选择位
```

因此，在 C51 语言可以直接对这些位进行操作，示例如下：

```
TR2=1                                       //启动定时器/计数器 T2
```

该语句直接置位 TR2，用于启动定时器/计数器 T2。

2. 控制寄存器 T2MOD

控制寄存器 T2MOD 可以选择 T2 是加 1 计数还是减 1 计数。控制寄存器 T2MOD 的格式及相关位的含义，如图 17-11 所示。

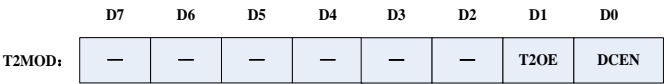


图 17-11 T2MOD 的格式定义

- D7~D0：保留位，做其他用途。

- T2OE: T2 输出启动位。
- DCEN: 置位为 1 时, 允许 T2 加 1 计数或减 1 计数。

T2MOD 寄存器的字节地址为 C9H, 不可位寻址。当 DCEN 复位为 0 时, 设置 T2 为加 1 计数方式; 当 DCEN 置位为 1 时, 由 T2EX (P1.1) 引脚上的逻辑电平, 设置 T2 为加 1 或减 1 计数方式。例如, 在 C51 语言中, 可采用如下语句设置 T2MOD。

```
sfr T2MOD=0xC9;  
T2MOD=0x00;
```

上面的语句首先定义 T2MOD 指向地址 0xC9, 接着赋值 0x00 将 T2 设置为加 1 计数方式。

17.3.2 定时器/计数器 2 的工作模式

定时器/计数器 2 是一个 16 位内部定时或对外部事件计数的计数器。T2 的工作模式与 T0 和 T1 有所不同。T2 具有三种工作模式, 即捕获、自动再装入和波特率发生器方式。通过软件设置控制寄存器 T2CON 的相关位, 可以设置 T2 的工作模式。位的设置和工作模式的对应关系如表 17-2 所示。

表 17-2 T2CON 的控制位和 T2 工作模式的对应关系

RCLK+TCLK	CP/ $\overline{\text{RL2}}$	TR2	工作模式
0	0	1	16 位自动再装入
0	1	1	16 位捕获
1	-	1	波特率发生器
-	-	0	工作停止

1. T2 的捕获模式

T2 的捕获模式即捕捉输入信号的相关信息, 常用于精确测量输入信号的变化。T2 的捕获是由寄存器 RCAP2H 和 RCAP2L 来控制。T2 工作在捕获模式下的组成结构, 如图 17-12 所示。

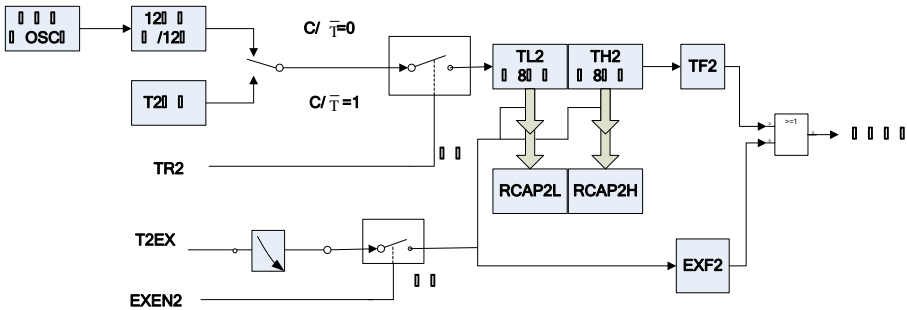


图 17-12 T2 捕获模式的组成结构

- 当 EXEN2=0 时, T2 为 16 位定时/计数器, 其工作情况和时序关系与 T0 和 T1 相同。

控制位 $C/\overline{\text{T2}}$ 用于选择 T2 是工作在计数器方式, 还是定时器方式。当 $C/\overline{\text{T2}}=0$ 时, TH2 和 TL2 计的是机器周期数, T2 作为定时器使用; 当 $C/\overline{\text{T2}}=1$ 时, 计数脉冲来自 T2 (P1.1) 引脚, T2 作为外部信号计数器使用。

- 当 EXEN2=1 时, T2 除了具有 16 位定时/计数器的功能外, 还具有捕获功能。

控制位 CP/ $\overline{\text{RL2}}$ (D0) 用来设置工作于捕获方式还是重新再装入方式。当 CP/ $\overline{\text{RL2}}=1$ 时, 选择捕获方式, 当 T2 计数满回 0 溢出或 T2EX (P1.1) 引脚上的负跳变, 都将选通三态门 (双向缓冲器), 触发捕获操作, 将 TH2 和 TL2 的内容传递给 RCAP2H 和 RCAP2L。当 RCLK=1 或 TCLK=1 时, CP/ $\overline{\text{RL2}}$ 标志位不起作用。T2 的计数满回 0 溢出时, 将强制 T2 进行捕获操作。

RCAP2L 和 RCAP2H 为分别为陷阱寄存器的低字节和高字节,其字节地址分别为 CAH 和 CBH。

T2EXN (P1.1) 引脚上的负跳变将置位 EXF2 标志位,向 CPU 申请中断。中断请求标志位须由软件置位或复位。

下面举例讲解其在程序设计中的应用,程序示例如下:

```
#include "reg52.h"                                //头文件

void Timer2_Server(void) interrupt 5
{
    EXF2=0;                                         //清零
    P2=~P2;                                         //反向
}

void main (void)                                   //主函数
{
    T2CON=0x0F;                                     //设置 T2
    TH2=0x0FF;                                     //初值
    TL2=0x0FE;
    ET2=1;                                          //允许 T2 定时器中断
    EA=1;                                          //打开总中断
    TR2=1;                                          //启动 T2 定时器

    while(1)                                       //主循环
    {
    }
}
```

在该程序中,首先设置 T2 并赋初值,接着打开中断。当外部 P1.1 引脚有负的电平跳变时将进入中断服务例程,反向 P2 端口。该程序可以在 Keil μ Vision3 中编译仿真运行。

2. T2 的重新再装入模式

当控制寄存器 T2MOD 的 DCEN 位设置为 0 时,T2 为加 1 再装入方式。此时根据控制寄存器 T2CON 中的 EXEN2 位,T2 可设置为不同的工作方式。分别介绍如下:

- 当 EXEN2=0 时,当 T2 计满回 0 溢出,将中断申请标志位 TF2 置位为 1,同时又将 RCAP2L、RCAP2H 中的初值分别重新再装入 TL2 和 TH2 中,继续下一轮计数。T2 的重新再装入模式的结构,如图 17-13 所示。这种工作方式的功能与 T0、T1 的模式 2 相同,只是 T2 为 16 位,计数范围更大。

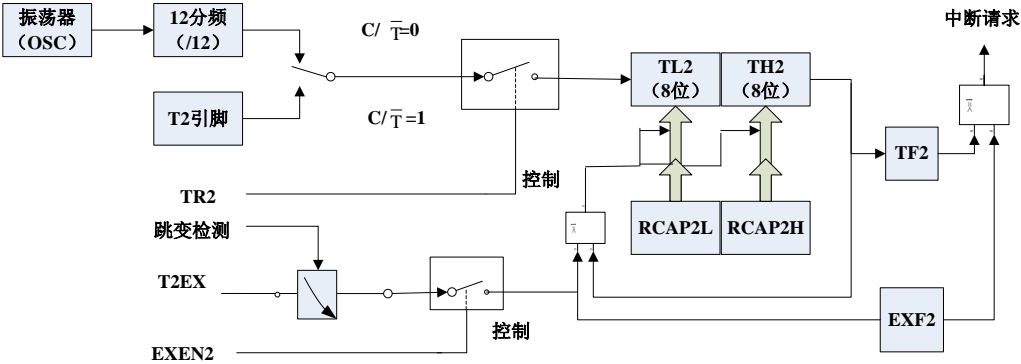


图 17-13 T2 重新再装入模式的结构

➤ 当 EXEN2=1 时, T2 除了上述功能外, 还具有新的功能。T2EXN (P1.1) 引脚上的负跳变将选通三态门(双向缓冲器), 从而触发重新再装入操作, 将 RCAP2H 和 RCAP2L 中的计数初值传送给 TH2 和 TL2, 并置位 EXF2 标志位为 1, 向 CPU 申请中断。

当控制寄存器 T2MOD 的 DCEN 位被置位为 1 时, 由 T2EX (P1.1) 引脚上的逻辑电平, 设置 T2 为加 1 或减 1 计数方式。其计数方式结构示意图, 如图 17-14 所示。

T2 在加 1/减 1 计数方式下, 当 T2EX=1 时, T2 执行加 1 计数方式, 不断加 1 计数向上溢出, 则置位 TF2 为 1, 并向 CPU 申请中断, 同时将 RCAP2L、RCAP2H 中的计数初值重新装入 TL2、TH2, 继续加 1 计数; 当 T2EX=0 时, T2 执行减 1 计数方式, 当 TL2、TH2 中的内容等于 RCAP2L、RCAP2H 中的计数初值时, 向下溢出, 则置位 TF2 为 1, 并向 CPU 申请中断, 同时将 0FFFH 重新装入 TL2、TH2, 继续减 1 计数。

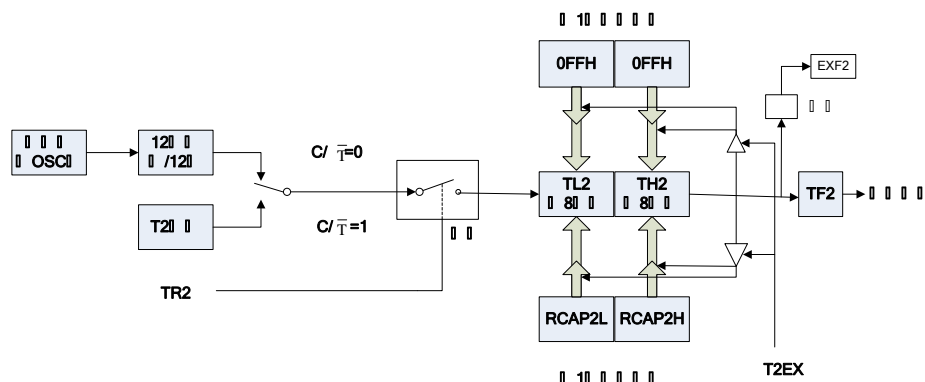


图 17-14 T2 加 1、减 1 计数方式的结构

无论向上还是向下溢出, EXF2 都会跳转并作为 17 分频位使用, 且 EXF2 不产生中断请求。中断标志位 TF2 和 EXF2 须由软件清 0。

下面举例讲解 T2 的重新再装入模式的应用。假设使用 AT89S52 单片机, 外接 12MHz 晶振, 使用定时器 T2 使 P1.0 引脚每个 1 秒改变一下输出电平, 即输出 0.5Hz 的方波脉冲。这里采用 C51 语言进行编写, 程序示例如下:

```
#include "reg52.h" //头文件
#define uint unsigned int
#define uchar unsigned char
sbit P1_0 = P1 ^ 0; //定义位变量

void Timer2_Server(void) interrupt 5
{
    static uint Timer2_Server_Count; //定义静态变量, 计数 T2 定时器的溢出次数

    TF2=0; //T2 定时器发生溢出中断时清除溢出标记

    Timer2_Server_Count++;
    if(Timer2_Server_Count==16) //T2 定时器的预装载值为 0x0BDC, 溢出 16
    //次就是 1 秒钟
    {
        Timer2_Server_Count=0;
        P1_0=~P1_0; //P1_0 反向
    }
}
```

```
void main (void)                                     //主函数
{
    P1_0=1;                                           //P1_0 置为低电平
    TH2=0x0B;                                         //T2 定时器赋预装载值，溢出 16 次就是 1 秒
    TL2=0xDC;
    RCAP2H=0x0B;                                     //重载值
    RCAP2L=0xDC;
    ET2=1;                                           //允许 T2 定时器中断
    EA=1;                                             //打开总中断
    TR2=1;                                           //启动 T2 定时器

    while(1)                                         //主循环
    {
    }
}
```

该程序可以在 Keil μ Vision3 中编译仿真,并可以下载到前面的 AT89S52 最小系统中执行。由于 T2 定时器是一个 16 位定时器,最长的溢出时间也就几十毫秒。因此在该程序中,为 T2 赋初值,使 T2 溢出 16 次的时间为 1 秒。程序中设置定时器 T2,并打开相应的中断,在中断服务例程中实现 P1.0 端口的方波输出。

3. T2 的波特率发生器模式

T2 的波特率发生器模式用于设置串行口的数据传送速率。控制寄存器 T2CON 中的 RCLK 和 TCLK 位,用于选择波特率发生器模式。当 RCLK 和 TCLK 中任何一个置位为 1 时,则串行通信进行接收/发送工作,T1 或 T2 工作于波特率发生器模式。

当 RCLK=0 且 TCLK=0 时,T1 工作于波特率发生器模式;当 RCLK=1 或 TCLK=1 时,T2 工作于波特率发生器模式,其组成结构如图 17-15 所示。其中 RCLK 选择接受波特率发生器,TCLK 选择发送波特率发生器。在波特率发生器模式,TL2、TH2 的内容不能再读写,也不能改写 RCAP2L、RCAP2H 的内容。

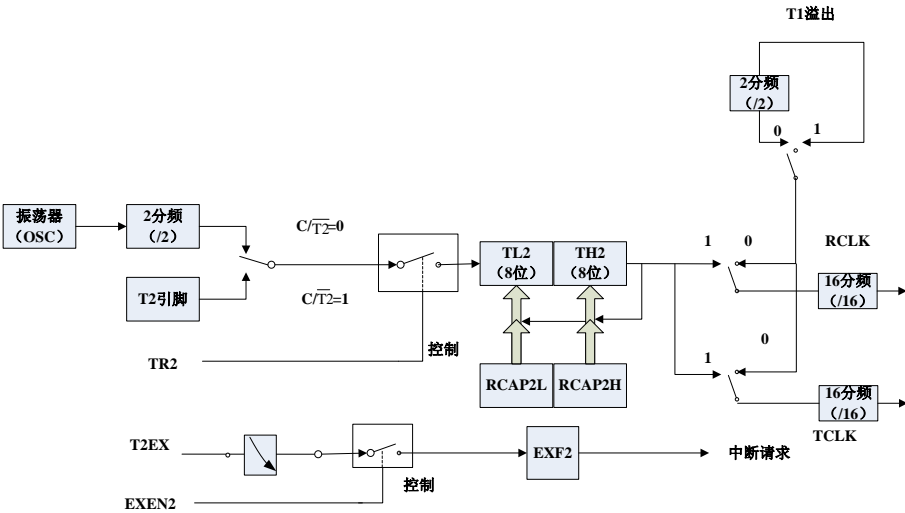


图 17-15 T2 工作于波特率发生器模式

当 T2 处于波特率发生器模式时,若 EXEN2=1,则 T2EX 端信号产生负跳变,EXF2 将置位为 1,但不发生捕获或重新装载操作。此时 T2EX 可作为一个外部中断源。

当 T2 处于波特率发生器模式时，其溢出脉冲用做串行口的时钟，此时钟频率可由内部时钟决定，也可由外部时钟决定。若 $C/\overline{T2}=1$ ，选用外部时钟作为计数脉冲，该时钟由 T2CLK 端输入，当外部脉冲出现负跳变时，计数器加 1，外部脉冲频率不超过振荡器频率的 1/24。由于溢出时，RCAP2L、RCAP2H 中的计数初值将重新装入 TL2、TH2 中，故波特率值还决定于装载值。若 $C/\overline{T2}=0$ ，选择内部时钟，计数脉冲频率为振荡频率的 1/2。

下面举例讲解如何在程序设计中使 T2 的波特率发生模式。这里采用 C51 语言进行设计，其程序代码示例如下：

```
#include <reg52.h>                                //头文件
sfr T2MOD=0xC9;

void SISR(void) interrupt 5                          //T2 中断响应
{
    TI=0;                                           //TI 清零
}

void main(void)                                     //主函数
{
    int a, bps = 96;                               //9600 bps
    SCON |= 0x50;                                   //串口方式 1，允许接收
    a=3456/bps;
    a=~a;
    RCAP2H = a/256;                                 //初值
    RCAP2L = a;
    T2MOD=0x00;                                     //加 1 计数方式
    T2CON=0x34;                                     //设置 T2 工作方式
    EA =1;                                          //开中断
    ES = 1;
    SBUF=0x67;
    while(1)                                       //主循环
    {
    }
}
```

在该程序中，首先初始化串口，并根据波特率设置 RCAP2H 和 RCAP2L 的初值，接着设置 T2 的工作方式并打开相应的中断。程序中将数据通过串口输出。该程序可以在 Keil μ Vision3 中编译仿真。

17.4 小结

本章详细讲述了 51 系列单片机的定时器/计数器的结构、控制寄存器及 4 种工作方式，并分别给出了详细的程序设计方法；本章还对 52 子系列的单片机定时器/计数器 T2 也进行了详细的介绍。定时器/计数器是单片机的一个非常有用的功能，熟练掌握本章内容，对读者以后的单片机设计有很大帮助。

第 18 章 51 系列单片机中断系统及其程序设计

在一般情况下，单片机程序是按照先后顺序执行代码的。有的时候，为了优先处理特定的任务，需要打断主程序的运行。这便需要单片机具有处理中断的能力。当微处理器的 CPU 正在执行程序的过程中，如果外部硬件或内部组件有紧急的请求，此时中断系统可以将当前的程序暂停，优先处理中断请求。当中断请求处理完毕的时候，再返回来继续执行主程序。

中断系统是计算机或单片机的重要功能部件，有了中断系统，便可以使微处理器具备对外部的异步事件进行处理的能力。51 系列单片机具有 5 个中断源，两级中断优先级，具有完善的指令控制能力。使用单片机的中断系统可以很方便地完成各种外部硬件响应的操作。

本章将主要介绍中断系统地概述，51 系列单片机的中断的类型、中断系统结构及控制寄存器；接着介绍了 CPU 的中断处理过程，并通过实例详细介绍了中断的程序设计；最后还将介绍 51 系列单片机外部中断源的扩展。

18.1 中断系统概述

中断系统是微处理器都具备的功能，不只是 51 系列单片机所特有的。不同的微处理器对中断系统支持的程度不同。中断其实是现实世界的一种反映。例如，某人正在看书，另外烧水的水壶开了，这个人便要去将水壶放好，等倒完水后再回来看书。这个过程便是“执行主程序（看书）—中断（水壶开，倒水）—中断返回（继续看书）”的过程。

18.1.1 什么是中断

微处理器引入中断后，便形成了一些其特有的概念，主要有如下几个。

- 中断源：能够产生中断请求的硬件或软件资源称为中断源；
- 主程序：中断发生前正在执行的程序代码称为主程序；
- 断点：中断发生时，主程序被断开的程序代码位置称为断点；
- 中断系统：在微处理器中，能够实现中断响应、中断处理和中断返回的功能部件称为中断系统。

利用这些基本概念来描述中断及其处理过程，即微处理器执行主程序，此时如果某个中断源产生中断请求，并发送给中断系统。中断系统响应该中断请求，并在断点处中断主程序的执行转而执行中断服务例程。当中断服务结束后，中断系统返回主程序的断点处，继续执行向下主程序。

18.1.2 中断的用途

在实际的测控应用系统中，中断的使用主要体现在如下几个方面。

1. 实时信号处理

微处理器的 CPU 对外部信号的检测可以采用查询和中断两种方式。其中查询方式是在主程序执行的过程中,按照一定的时间间隔循环扫描端口,根据端口电平的改变从而获知外部事件的发生。由于扫描端口是有一定时间间隔的,因此有可能会错过对事件的响应。

如果采用中断法,则当外部事件发生的时候,立刻向 CPU 发送中断请求,CPU 便会立刻响应,并加以处理。中断法的响应比较快,适合于实时性要求比较高的场合。

2. 系统故障处理

在系统运行过程中,如果遇到电源突变、计算溢出等,则可能导致无法预测的硬件及软件方面的故障。使用以往的方法,需要关闭整个系统,然后进行故障处理。此时有些系统执行过程中的参数便无法保存,有可能导致不可挽救的损失。

如果使用相应的中断请求,则在系统故障发生的时候,立即进行相应的中断处理,保护重要的参数并进行系统的自动修复。这样系统的抗干扰性和可维护性大大提高。

中断系统是一个非常实用的微处理器组件,合理使用中断技术,可以极大地提高 CPU 的工作效率和微处理器解决问题的能力。

18.1.3 中断需要解决的问题

一个完整的中断系统需要合理处理中断识别、中断响应、中断优先级及中断嵌套等。下面分别进行介绍。

1. 中断识别

中断识别是指微处理器对各个中断的区分。一个微处理器往往不止支持一个中断,例如,8051 单片机支持 5 个中断源,而 AT89S52 则支持多达 8 个中断源。这些中断需要对应不同的中断服务程序,因此,中断系统必须能够对各个不同的中断源进行区分。

在微处理器中,一般采用中断矢量的方法来识别中断。在硬件系统中为不同的中断源分配不同的中断服务程序入口地址,即向量地址。当某个中断发生的时候,程序跳转向指定的中断服务程序入口地址开始执行程序,这样便实现了对不同中断的识别及响应。

2. 中断响应及返回

中断响应是中断发生后所需要处理的工作,而中断返回则是中断处理结束后所需要处理的工作。

一个典型的中断处理的流程,如图 18-1 所示。在主程序执行过程中,如果一个中断源产生中断请求,中断系统应该能中止当前程序的执行,并且保护程序当前的各种参数,即保护现场。中断正确响应后,便进入中断服务程序,执行相应的处理过程。中断执行完毕后,中断系统负责返回到主程序的断点处,并恢复现场,继续执行断点以下的主程序。

3. 中断优先级及中断嵌套

微处理器的中断系统一般都支持多个中断源,此时经常遇到有两个中断同时提出请求,以及在一个中断得到请求并进行处理的进程中,另外一个中断请求随之发生并需要得到处理的情况。为了有效地解决这两个问题,在微处理器的中断系统中提出了中断优先级的概念。

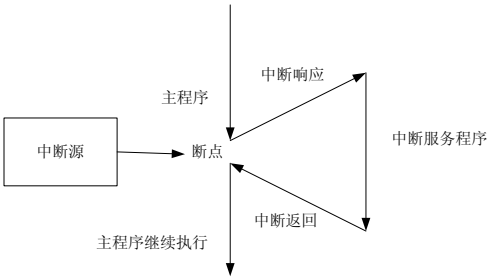


图 18-1 中断处理流程

在微处理器的中断系统中，每个中断源赋予不同的优先级，根据优先级的不同来执行中断请求。这样对于上面的两个问题，便可以使用中断优先级排队和中断嵌套来解决。解决的原则如下：

- 在同一时刻，有两个中断同时提出请求时，中断系统按照中断源优先级的高低进行逐次响应。即优先级高的中断优先处理，处理完毕后，再处理优先级低的中断。这个过程称为中断优先级排队。
- 在一个中断得到请求并进行处理的进程中，如果另外一个中断发生，则判断其优先级的高低。如果新中断的优先级高于原中断，则在原中断服务程序中产生新断点，并转而执行新的中断服务子程序，然后逐级返回。这个过程称为中断嵌套。如果新中断的优先级低于原中断，而继续保持原中断的执行，直至完毕后返回再执行新中断请求。

51 系列单片机的中断系统支持两级中断优先级，并可以实现两级中断嵌套。中断嵌套的示意图，如图 18-2 所示。

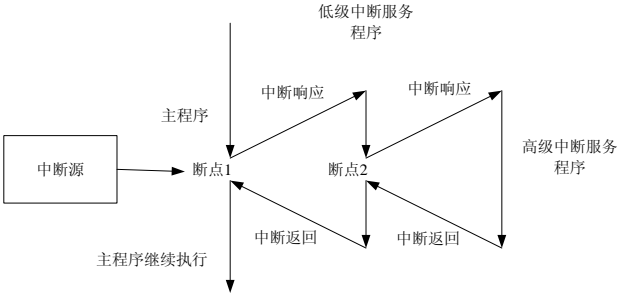


图 18-2 中断嵌套示意图

这里需要注意的是，中断服务的过程和子程序的调用有点相似，但是两者是不同的概念，主要体现在如下几个方面。

- 源不同。中断是外部中断源产生的，具有时间上的不可预测性，因此中断服务程序的调用是随机的；而子程序调用则是主程序事先安排好的，程序执行到特定位置便进行子程序调用。
- 响应不同。中断发生的时候，中断系统需要进行保护断点和保护现场等操作；而调用子程序的时候只保护当前断点。
- 功能不同。中断服务程序主要是处理 CPU 外部的异步事件；而子程序调用则主要为子程序服务，实现特定的功能。

18.2 51 系列单片机的中断类型

微处理器一般都支持多个中断源，通常按照中断源的不同，大致可以分为三类，即外部中

断源、定时中断源和串行中断源。以 8051 单片机为例，其共有 5 个中断源，包括两个外部中断源、两个定时中断源和一个串行中断源。下面分别介绍这三类中断源。

18.2.1 外部中断源

外部中断源是由外部硬件电路产生的中断。外部中断源一般采用两种触发方式，即沿触发方式（上升沿或下降沿有效）和电平触发方式（高电平或低电平有效）。下面分别进行介绍。

- 对于沿触发方式，在微处理器的时钟周期内如果检测到有效的电平变化信号（上升沿或下降沿），则即认为产生中断请求，CPU 开始执行相应的中断服务程序。
- 对于电平触发方式，在微处理器的时钟周期内如果检测到有效的电平信号（高电平或低电平），即认为产生中断请求，CPU 开始执行相应的中断服务程序。

51 系列单片机提供了两个外部中断源，分别为 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ ，分别介绍如下：

- $\overline{\text{INT0}}$ （P3.2 引脚）为外部中断 0 请求。其中断触发方式可以选择，当置 $\text{TCON}.0=0$ 时，低电平有效；当置 $\text{TCON}.0=1$ 时，下降沿有效。当向 CPU 申请中断成功后，将建立 IE0 标志。
- $\overline{\text{INT1}}$ （P3.3 引脚）为外部中断 1 请求。其中断触发方式可以选择，当置 $\text{TCON}.2=0$ 时，低电平有效；当置 $\text{TCON}.2=1$ 时，下降沿有效。当向 CPU 申请中断成功后，将建立 IE1 标志。

18.2.2 定时中断源

定时中断源是由微处理器内部集成的定时器/计数器产生的中断。其以定时器/计数器的溢出信号作为中断请求，当溢出发生的时候，CPU 响应中断请求并开始执行相应的中断服务程序。

51 系列单片机内部集成了两个定时器/计数器，分别提供了 TF0 和 TF1 两个定时中断源。分别介绍如下：

- TF0 （ $\text{TCON}.5$ ）为定时器/计数器 T0 的溢出中断请求。在程序运行时，如果定时器/计数器 T0 产生溢出，将标志位 TF0 置位，请求中断，中断系统将进入中断处理。
- TF1 （ $\text{TCON}.7$ ）为定时器/计数器 T1 的溢出中断请求。在程序运行时，如果定时器/计数器 T1 产生溢出，将标志位 TF1 置位，请求中断，中断系统将进入中断处理。

18.2.3 串行中断源

串行中断源是微处理器在进行串行通信的时候，每个数据发送或接收完毕产生的中断请求。其来自于微处理器的内部。CPU 响应这个中断请求后，可以再次发送串行数据或读入接收到的串行数据。

51 系列单片机的片内串口接口提供了发送中断源 TI 和接收中断源 RI 。当进行串行数据传输的时候，如果发送或接收串行数据完成的时候，硬件将置标志位 TI 或 RI ，请求中断，中断系统将进入中断处理。

51 系列单片机中断源的识别和响应是通过中断入口地址，即中断矢量来实现的。当某个中断发生的时候，首先 CPU 响应中断，将中断源的入口地址装载到程序计数器 PC 中，中断服务程序便是从该地址开始执行的。当中断服务程序执行完毕后，将返回原来的断点。

18.3 51 系列单片机的中断系统

51 系列单片机在硬件上提供了 5 个中断源支持，在软件上可以使用控制寄存器来进行控制，为用户提供了十分方便的操作接口。下面分别介绍中断请求、中断允许及中断优先级的标志及访问。

18.3.1 中断请求标志及其访问

中断请求标志用于中断发生时，向 CPU 提出请求。51 系列单片机的 5 个中断请求，分别对应一个中断请求标志位。其中特殊功能寄存器 TCON 控制外部中断和定时器溢出中断，特殊功能寄存器 SCON 控制串行接口中断。下面分别进行介绍。

1. TCON 的中断标志

控制寄存器 TCON 的位定义格式，如图 18-3 所示。控制寄存器 TCON 的字节地址 88H，其中的每一位都是可以进行位寻址的，位地址为 88H~8FH。下面分别介绍各位的定义。

	D7	D6	D5	D4	D3	D2	D1	D0	可位寻址
位地址	8F	8E	8D	8C	8B	8A	89	88	
位符号	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	

图 18-3 控制寄存器 TCON 的格式

- **TF1**：定时器/计数器 T1 的溢出标志位。当定时器/计数器 T1 产生溢出时，单片机将自动置 TF1=1。此时，CPU 响应中断，转向相应的中断服务程序，并自动置 TF1=0。
- **TF0**：定时器/计数器 T0 的溢出标志位。当定时器/计数器 T0 产生溢出时，单片机将自动置 TF0=1。此时，CPU 响应中断，转向相应的中断服务程序，并自动置 TF0=0。
- **IE1**：外部中断 1 请求标志位。当单片机 $\overline{\text{INT1}}$ 端口的中断信号有效的时候，单片机将自动置 IE1=1 请求中断。CPU 响应中断请求，转向对应的中断服务程序，并自动置 IE1=0。
- **IT1**：外部中断 1 的中断触发方式控制位。当 IT1=0 的时候，为低电平触发方式；当 IT1=1 的时候，为下降沿触发方式。
- **IE0**：外部中断 0 请求标志位。当单片机 $\overline{\text{INT0}}$ 端口的中断信号有效的时候，单片机将自动置 IE0=1 请求中断。CPU 响应中断请求，转向对应的中断服务程序，并自动置 IE0=0。
- **IT0**：外部中断 0 的中断触发方式控制位。当 IT0=0 的时候，为低电平触发方式；当 IT0=1 的时候，为下降沿触发方式。

控制寄存器 TCON 的 D4 和 D6 位为 TR0 和 TR1，用于控制定时器/计数器的启动停止，可以参阅上一章的介绍。在汇编语言程序中，可以采用如下语句设置 TCON 寄存器。

```
SETB      IT0                                ;下降沿触发方式
```

该语句采用位寻址的方式，置位 IT0，用于设置外部中断 0 为下降沿触发方式。在 C51 语言中，头文件 REG51.H 包含了 TCON 的位定义，示例如下：

```
sbit      TF1  = 0x8F;                        //溢出标志位 TF1
sbit      TR1  = 0x8E;                        //启/停控制位 TR1
sbit      TF0  = 0x8D;                        //溢出标志位 TF0
sbit      TR0  = 0x8C;                        //启/停控制位 TR0
sbit      IE1  = 0x8B;                        //外部中断 1 请求标志位
sbit      IT1  = 0x8A;                        //外部中断 1 的中断触发方式控制位
sbit      IE0  = 0x89;                        //外部中断 0 请求标志位
sbit      IT0  = 0x88;                        //外部中断 0 的中断触发方式控制位
```

因此，在 C51 语言可以直接对这些位进行操作，示例如下：

```
IT0=1                                          //下降沿触发方式
```

该语句直接置位 TR0，用于设置外部中断 0 为下降沿触发方式。

2. SCON 的中断标志

控制寄存器 SCON 的位定义格式，如图 18-4 所示。控制寄存器 SCON 的字节地址 98H，其中的每一位都是可以进行位寻址的，位地址为 98H~9FH。下面分别介绍各位的定义。

	D7	D6	D5	D4	D3	D2	D1	D0	可位寻址
位地址	9F	9E	9D	9C	9B	9A	99	98	
位符号	—	—	—	—	—	—	TI	RI	

图 18-4 控制寄存器 SCON 的格式

- **RI**：串行接口接收数据中断请求标志位。当单片机的串行接口接收完一个数据后，硬件自动置 RI=1。此时，CPU 响应中断，转向相应的中断服务程序。注意此时不会自动 RI 清零，须在软件中置 RI=0。
- **TI**：串行接口发送数据中断请求标志位。当单片机的串行接口发送完一个数据后，硬件自动置 TI=1。此时，CPU 响应中断，转向相应的中断服务程序。注意此时不会自动 TI 清零，须在软件中置 TI=0。

控制寄存器 SCON 用于其他用途，这里不做具体介绍。

由于控制寄存器 SCON 可以位操作，因此在汇编语言程序中可以采用如下语句设置 SCON 寄存器。

```
CLR      RI                      ;RI 清零
```

该语句采用位寻址的方式，清零 RI，用于清除串行接口接收数据中断。在 C51 语言中，头文件 REG51.H 包含了 SCON 的位定义，示例如下：

```
sbit      SM0  = 0x9F;
sbit      SM1  = 0x9E;
sbit      SM2  = 0x9D;
sbit      REN  = 0x9C;
sbit      TB8  = 0x9B;
sbit      RB8  = 0x9A;
sbit      TI   = 0x99;           //串行接口发送数据中断请求标志位
sbit      RI   = 0x98;           //串行接口接收数据中断请求标志位
```

因此，在 C51 语言可以直接对这些位进行操作，示例如下：

```
RI=0                                //下降沿触发方式
```

该语句直接赋值 RI 为 0，用于清除串行接口接收数据中断。

18.3.2 中断允许标志及其访问

中断允许标志用于允许或禁止相应的中断请求。对于 51 系列单片机的 5 个中断请求，分别对应一个中断允许或禁止标志位，其均由一个中断允许控制寄存器 IE 控制。

中断允许控制寄存器 IE 的位定义格式，如图 18-5 所示。其字节地址为 A8H，可以位寻址。下面分别介绍各位的定义。

- **EA**：中断允许或禁止总控制位。当置 EA=0 时，单片机将禁止所有中断，不响应任何中断请求；当置 EA=1 时，单片机允许各个中断，此时还需要由其他标志位确定各个中断的允许或禁止。

	D7	D6	D5	D4	D3	D2	D1	D0	可位寻址
位地址	AF	AE	AD	AC	AB	AA	A9	A8	
位符号	EA	—	—	ES	ET1	EX1	ET0	EX0	

图 18-5 控制寄存器 IE 的格式

- **ES**: 串行中断允许或禁止控制位。当置 ES=0 时, 禁止串行口中断; 当置 ES=1 时, 允许串行口中断。
- **ET1**: 定时器/计数器 T1 允许或禁止标志位。当置 ET1=0 时, 禁止定时器/计数器 T1 中断; 当置 ET1=1 时, 允许定时器/计数器 T1 中断。
- **EX1**: 外部中断 1 允许或禁止标志位。当置 EX1=0 时, 禁止外部中断 1; 当置 EX1=1 时, 允许外部中断 1。
- **ET0**: 定时器/计数器 T0 允许或禁止标志位。当置 ET0=0 时, 禁止定时器/计数器 T0 中断; 当置 ET0=1 时, 允许定时器/计数器 T0 中断。
- **EX0**: 外部中断 0 允许或禁止标志位。当置 EX0=0 时, 禁止外部中断 0; 当置 EX0=1 时, 允许外部中断 0。

中断允许控制寄存器 IE 的 D5、D6 位未定义, 不使用。由于寄存器 IE 可以位寻址, 因此在汇编语言程序中, 可以采用如下语句设置 IE 寄存器。

```
SETB    EA                                ; 允许所有中断
```

该语句采用位寻址的方式, 置位 EA, 用于允许所有中断。在 C51 语言中, 头文件 REG51.H 包含了 IE 的位定义, 示例如下:

```
sfr      IE   = 0xA8;                    // 定义 IE
sbit     EA   = 0xAF;                    // 中断允许或禁止总控制位
sbit     ES   = 0xAC;                    // 串行中断允许或禁止控制位
sbit     ET1  = 0xAB;                    // 定时器/计数器 T1 允许或禁止标志位
sbit     EX1  = 0xAA;                    // 外部中断 1 允许或禁止标志位
sbit     ET0  = 0xA9;                    // 定时器/计数器 T0 允许或禁止标志位
sbit     EX0  = 0xA8;                    // 外部中断 0 允许或禁止标志位
```

因此, 在 C51 语言可以直接对这些位进行操作, 示例如下:

```
EA=1                                       // 允许所有中断
```

该语句直接为 EA 赋值 1, 用于允许所有中断。

18.3.3 中断优先级标志及其访问

中断优先级标志用于设置中断的优先等级。对于 51 系列单片机支持两级中断优先级, 其 5 个中断请求, 分别对应一个中断优先级标志位, 由寄存器 IP 来控制。

中断优先级控制寄存器 IP 各位的定义, 如图 18-6 所示。其字节地址为 B8H, 可以位寻址。下面分别介绍各位的定义。

	D7	D6	D5	D4	D3	D2	D1	D0	可位寻址
位地址	BF	BE	BD	BC	BB	BA	B9	B8	
位符号	—	—	—	PS	PT1	PX1	PT0	PX0	

图 18-6 控制寄存器 IP 的格式

- **PS**: 串行接口中断优先级设置位。当置 PS=0 时, 该中断源被定义为低优先级; 当置 PS=1 时, 该中断源被定义为高优先级。
- **PT1**: 定时器/计数器 T1 优先级设置位。当置 PT1=0 时, 该中断源被定义为低优先级; 当置 PT1=1 时, 该中断源被定义为高优先级。
- **PX1**: 外部中断 1 优先级设置位。当置 PX1=0 时, 该中断源被定义为低优先级; 当置 PX1=1 时, 该中断源被定义为高优先级。
- **PT0**: 定时器/计数器 T0 优先级设置位。当置 PT0=0 时, 该中断源被定义为低优先级;

当置 PT0=1 时，该中断源被定义为高优先级。

- **PX0**：外部中断 0 优先级设置位。当置 PX0=0 时，该中断源被定义为低优先级；当置 PX0=1 时，该中断源被定义为高优先级。

中断优先级控制寄存器 IP 的 D5~D7 位未定义，不使用。

由于寄存器 IE 可以位寻址，因此在汇编语言程序中，可以采用如下语句设置 IE 寄存器。

```
SETB    PS                                ; 串行接口中断源为高优先级
```

该语句采用位寻址的方式，置位 PS，用于设置串行接口中断源为高优先级。在 C51 语言中，头文件 REG51.H 包含了 IP 的位定义，示例如下：

```
sfr      IP   = 0xB8;                    // 定义 IP
sbit     PS   = 0xBC;                    // 串行接口中断优先级设置位
sbit     PT1  = 0xBB;                    // 定时器/计数器 T1 优先级设置位
sbit     PX1  = 0xBA;                    // 外部中断 1 优先级设置位
sbit     PT0  = 0xB9;                    // 定时器/计数器 T0 优先级设置位
sbit     PX0  = 0xB8;                    // 外部中断 0 优先级设置位
```

因此，在 C51 语言可以直接对这些位进行操作，示例如下：

```
PS=1                                       // 串行接口中断源为高优先级
```

该语句直接为 PS 赋值 1，用于设置串行接口中断源为高优先级。

18.4 中断的处理过程

51 系列单片机对中断的处理分为三步，分别为中断响应，中断处理和中断返回。下面分别介绍这三个过程。

18.4.1 中断响应

中断响应是指在单片机主程序运行过程中，如果遇到中断请求，在满足中断响应条件的情况下，CPU 对该中断做出的响应。在实际的中断处理时，需要注意三个方面的问题，即中断响应的条件、中断响应的过程和中断响应的的时间，下面分别进行介绍。

1. 中断响应的条件

当某个中断源发出中断请求信号的时候，单片机 CPU 并不总是对该中断进行响应的。一般来说，单片机对中断的响应遵循如下几个原则：

- 单片机应允许所有中断源申请中断，即置中断允许总控制位 EA=1。这样 CPU 才可以响应中断。
- 为了允许某个独立中断源的中断请求，应置相应的允许位（PX0、PX1、PT0、PT1、PS）为 1。
- 如果多个同优先级的中断请求同时发出，则 CPU 按照一定的查询次序来决定中断执行的顺序。51 系列单片机对中断的查询次序为“外部中断 0→定时器/计数器 T0→外部中断 1→定时器/计数器 T1→串行接口中断”。
- 如果一个低优先级的中断请求正在执行，则可以中断该服务程序，然后执行本次中断请求。如果一个高优先级或同优先级的中断请求正在执行，则当前中断请求不会立即执行。
- 任何正在执行的指令在未完成前，中断请求都不会响应。
- 如果程序正在执行读写寄存器 IE 和 IP 指令，则执行完该命令后，需要再执行一条其他指令才可以响应中断。
- 如果程序正在执行返回指令，则执行完该命令后，需要再执行一条其他指令才可以响

应中断。

2. 中断响应的过程

当一个中断请求满足中断响应条件后，CPU 便可以执行中断响应。中断响应的执行包含如下几个步骤。

- (1) 单片机 CPU 响应中断，硬件自动将当前的断点地址压入堆栈。
 - (2) CPU 将相应的中断入口地址装入程序计数器 PC 中。
 - (3) 程序转向相应的中断入口地址，开始执行中断服务例程。
 - (4) 对于部分中断，进入中断服务例程后，硬件还将自动将中断标志位清零。
- 51 系列单片机为每个中断源分配一个入口地址，如表 18-1 所示。

表 18-1 中断源的入口地址

中 断 源	入口地址	中 断 源	入口地址
0 (外部中断 0)	0003H	16	0083H
1 (定时器/计数器 T0)	000BH	17	008BH
2 (外部中断 1)	0013H	18	0093H
3 (定时器/计数器 T1)	001BH	19	009BH
4 (串行接口中断)	0023H	20	00A3H
5 (定时器/计数器 T2)	002BH	21	00ABH
6 (PCA 中断)	0033H	22	00B3H
7	003BH	23	00BBH
8	0043H	24	00C3H
9	004BH	25	00CBH
10	0053H	26	00D3H
11	005BH	27	00DBH
12	0063H	28	00E3H
13	006BH	29	00EBH
14	0073H	30	00F3H
15	007BH	31	00FBH

从表 18-1 中可以看出，两个相邻的中断入口地址很接近，一般放置不了几个指令。实际程序设计中，一般都是将一个跳转指令放置在该处。当中断响应的时候，执行这个跳转指令，转向相应的中断服务子程序。

3. 中断响应的时间

中断响应的时间，即 CPU 从检查中断请求标志位 (TCON 或 SCON)，到转向对应的中断入口地址所需的机器周期个数。单片机 CPU 对中断的响应是需要一定时间的，即不是立刻进入真正的中断服务例程。对于实时性要求比较高的应用场合，就需要严格考虑中断的响应时间。

51 系列单片机对一个中断的正确响应，首先需要查询标志位 (占用一个机器周期)，然后产生 LCALL 指令 (两个机器周期)，转向中断入口地址。因此一个中断响应最少需要三个机器周期才能完成。

实际应用中，如果碰到不满足前面的中断响应条件的场合，则中断将被等待处理，此时便需要更长的响应时间。具体的时间需要根据等待指令的个数及周期数。

18.4.2 中断处理

单片机 CPU 对中断处理的流程图，如图 18-7 所示。从图 18-7 中可以看出，单片机 CPU 对中断处理的过程可以分为两类，一类是硬件自动完成的部分，另一类是软件处理的部分。这里主要介绍的中断处理是指中断的软件处理过程。

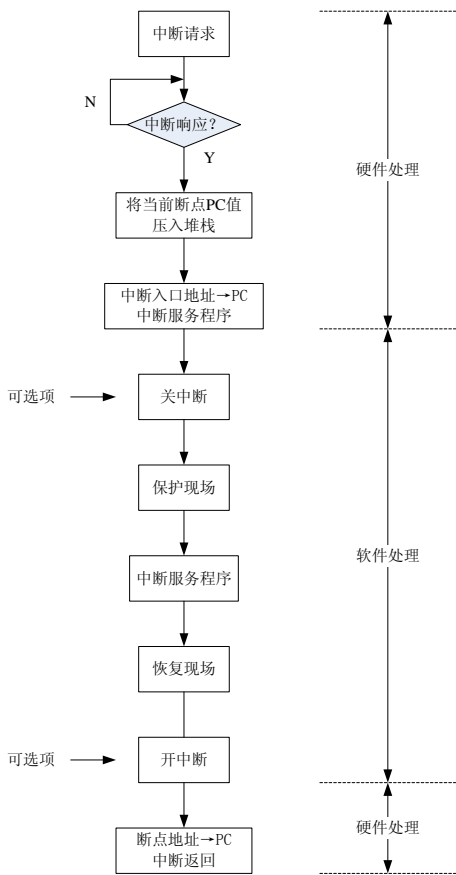


图 18-7 中断处理流程图

中断的软件处理过程是整个中断服务例程，即从程序计数器 PC 指向中断入口地址开始，到中断结束返回为止。这个中断服务程序一般需要完成以下 4 个部分的操作。

1. 程序跳转

由于相邻的两个中断其中断入口地址之间只相隔 8 个字节，一般放置不下一个完整的中断服务程序。因此，往往将中断服务程序放置在其他的地址空间，而在相应的中断入口地址处放置一个无条件转移指令，将程序转向对应的中断服务例程。

2. 关中断与开中断

如果该中断正在执行过程中，不想被更高级的中断打断，则在进入中断服务例程后可以置 EA=0，关闭所有中断，或者关闭某些中断。这样可以保证中断服务程序的顺利执行。在中断服务例程结束的时候，可以将关闭的中断开启，以便于能够接收新的中断请求。

3. 现场保护与恢复现场

一般来说，在主程序和中断服务程序中都会用到累加器、寄存器等。CPU 在中断服务程序中使用这些寄存器的时候将会改变其中的内容，再返回主程序的时候容易引起混乱。因此，在进入中断服务程序后应该首先将这些重要的寄存器保存，即保护现场；当中断服务程序结束前应该将这些寄存器的内容恢复，即现场恢复。

4. 处理中断源请求

中断服务程序主要是为了对特定的中断进行特定的处理。因此，还需要用户根据系统的需

要编写相应的中断处理请求程序。这是用户的特定任务代码段。

18.4.3 中断返回

中断返回是指中断服务例程结束后，返回主程序的过程。在 51 系列单片机中，中断服务程序的返回比较简单，直接在中断服务程序最后面加上一个中断返回指令 **RETI** 即可。中断返回指令 **RETI** 执行时，主要完成如下两个方面的工作。

- 将主程序的断点地址送回程序计数器 **PC** 中。
- 通知中断系统已将当前中断处理完毕，清除优先级状态触发器。

18.4.4 中断请求的撤离

中断请求的撤离主要的目的是保证对于一次中断信号只执行一次中断响应。**CPU** 响应中断后，需要及时将中断请求标志 **TCON** 或 **SCON** 中的标志位清除，否则将引起一个中断信号触发多次中断响应。

51 系列单片机可以采用两种方式解决中断请求的撤离，分别介绍如下：

- 硬件自动清除中断标志。对于外部中断 0、外部中断 1、定时器/计数器 **T0** 和 **T1** 来说，在 **CPU** 正确响应中断后，将自动清除该中断请求的标志位，无须软件处理。
- 软件清除中断标志。对于串行接口的中断请求，当 **CPU** 正确响应中断后，硬件不会自动清除中断标志位 **TI** 或 **RI**，因此需要在中断服务程序中用软件设置清除。

另外，对于外部中断，一般推荐采用沿触发方式，而不采用电平触发方式。因为电平触发方式，有可能触发电平在很长一段时间内都保持，这样容易引起再次触发。如果必须使用电平触发，则应该在硬件或软件上将输入的触发信号及时翻转过来，以防止同一个中断被响应多次。

18.5 中断源的程序设计

前面介绍了 51 系列单片机中的各个中断源及其程序设计。下面将分别介绍在程序设计中如何实现中断的响应及处理。

18.5.1 外部中断源的程序设计

51 系列单片机提供了两个外部中断源， $\overline{\text{INT0}}$ (**P3.2** 引脚) 和 $\overline{\text{INT1}}$ (**P3.3** 引脚)，分别对应外部中断 0 请求和外部中断 1 请求。下面举例讲解外部中断的程序设计。假设要使外部中断发生的时候，**P1** 端口和 **P2** 端口的电平分别反向。如果采用汇编语言进行程序设计，其程序示例如下：

```
ORG      0000H
AJMP START                      ; 转向主程序
ORG      0003H
AJMP ITRU0                      ; 转向外部中断 0 子程序
ORG      0013H
AJMP ITRU1                      ; 转向外部中断 1 子程序
ORG      0200H
START:   MOV      IP, #05H      ; 外部中断 0 和外部中断 1 设置为高优先级
        SETB IT0              ; 外部中断 0 为下降沿触发方式
        SETB IT1              ; 外部中断 1 为下降沿触发方式
        SETB EX0              ; 开 EX0 中断
        SETB EX1              ; 开 EX1 中断
        SETB EA               ; 开中断
HERE:    SJMP HERE
ORG      0400H
```



```
ITR0:  MOV      A,P1                ;外部中断 0 服务例程
      CPL      A                    ;反相
      MOV      P1,A
      RETI                          ;返回
ITR1:  MOV      A,P2                ;外部中断 1 服务例程
      CPL      A                    ;反相
      MOV      P2,A
      RETI                          ;返回
      END                          ;结束
```

在该程序中，分别在外部中断 0 和 1 的中断入口处放置无条件转移指令，将程序转向相应的中断服务例程。主程序中将外部中断 0 和外部中断 1 设置为高优先级，并设置外部中断 0 和 1 为下降沿触发方式。最后打开相应的中断，并置 EA=1 开总中断。当 $\overline{INT0}$ （P3.2 引脚）和 $\overline{INT1}$ （P3.3 引脚）有负跳变时，将触发相应的中断，分别置 P1 端口和 P2 端口反相。如果采用 C51 语言进行程序设计，其程序示例如下：

```
#include<reg51.h>                //头文件

void ISR0(void) interrupt 0        //外部中断 0 服务例程
{
    P1=~P1;                        //P1 端口反相
}

void ISR1(void) interrupt 2        //外部中断 1 服务例程
{
    P2=~P2;                        //P2 端口反相
}

void main(void)                   //主函数
{
    IP=0x05;                       //外部中断 0 和外部中断 1 设置为高优先级
    IT0=1;                          //外部中断 0 为下降沿触发
    IT1=1;                          //外部中断 1 为下降沿触发
    EX0=1;                          //开 EX0 中断
    EX1=1;                          //开 EX1 中断
    EA=1;                           //开总中断
    while(1)                        //主循环
    {
        ;
    }
}
```

在该程序中，分别定义了外部中断 0 和 1 的服务例程。在主函数中，将外部中断 0 和外部中断 1 设置为高优先级，并设置外部中断 0 和 1 为下降沿触发方式。最后打开相应的中断，并置 EA=1 开总中断。在主循环中不进行任何操作。当 $\overline{INT0}$ （P3.2 引脚）和 $\overline{INT1}$ （P3.3 引脚）有负跳变时，将触发相应的中断，分别置 P1 端口和 P2 端口反相。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

18.5.2 定时中断源的程序设计

51 系列单片机内部集成了两个定时器/计数器，分别提供了 TF0 和 TF1 两个定时中断源。下面举例讲解定时中断的程序设计。假设单片机外接 6MHz 晶振，这里采用定时器 T0，采用

51 单片机开发与应用技术详解

模式 0 使定时器产生 1ms 的定时，并在 P1 端口输出周期为 2ms 的方波。

- (1) 按照前面章节，计算单片机的机器周期为 $12 \div 6\text{MHz} = 2 \times 10^{-6}\text{s} = 2\mu\text{s}$ 。
- (2) 计算定时器初值 X，则 $(2^{13}-X) \times 2 \times 10^{-6} = 1 \times 10^{-3}$ 。
- (3) 求得 $X=7692=1\text{E}0\text{CH}$ 。
- (4) 根据 13 位定时器的结构特性，定时器 T0 的初值应该设置为 TH0=0F0H，TL0=0CH。如果采用汇编语句进行程序设计，其程序示例如下：

```

ORG      0000H
AJMP START                      ; 转向主程序
ORG      000BH
AJMP ITRU                       ; 转向中断子程序
START:   MOV      SP, #60H
MOV      TMOD, #00H             ; 设置定时器 0 模式 0
MOV      TL0, #0CH              ; 初始化
MOV      TH0, #0F0H
SETB TR0
SETB ET0
SETB EA                          ; 开中断
HERE:    SJMP HERE
ITRU:    MOV      TL0, #0CH      ; 重置初值
MOV      TH0, #0F0H
MOV      A, P1
CPL      A                      ; 反相
MOV      P1, A
RETI
END
```

上面的程序中采用了中断服务程序产生要求的方波，在中断服务子程序中，首先将定时器初值再次装入定时器，以便于开始下一次定时，然后通过取反 P1 端口得到方波波形。在主程序中，则设置定时器 0 为模式 0，并初始化定时初值，开启动定时器，开总中断。该程序在执行时，将周期性地置 P1 端口反相。如果采用 C51 语言进行程序设计，其程序示例如下：

```

#include <reg51.h>               // 头文件

void T0ISR(void) interrupt 1     // 定时器 T0 中断响应
{
    TL0=0x0C;                   // 重置计数初值
    TH0=0x0F0;
    P1=~P1;                     // 反相
}

void main(void)                 // 主函数
{
    TMOD=0x00;                  // 设置定时器 T0 为模式 0
    TL0=0x0C;                   // 初始化
    TH0=0x0F0;
    TR0=1;
    ET0=1;
    EA=1;                       // 开中断
    while(1)                   // 主循环
    {
    }
}
```

上面的程序中定义了定时器 0 的中断服务程序产生要求的方波，在中断服务子程序中，首先将定时器初值再次装入定时器，以便于开始下一次定时，然后通过取反 P1 端口得到方波波形。在主函数中，则设置定时器 0 为模式 0，并初始化定时初值，开启启动定时器，开总中断。该程序在执行时，定时器溢出时将周期性地触发中断，置 P1 端口反相。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

18.5.3 串行中断源的程序设计

51 系列单片机的片内串口接口提供了发送中断源 TI 和接收中断源 RI。下面举例讲解定时中断的程序设计。假设使用串行口的发送中断源 TI 来发送数据。如果采用汇编语句进行程序设计，其程序示例如下：

```

ORG      0000H
AJMP MAIN                ; 转向主程序
ORG      0023H
AJMP SISR                ; 转向中断服务例程
ORG      0200H
MAIN:    MOV      SCON, #00H        ; 初始化串行口模式 0
        SETB ES                    ; 开启串行中断
        MOV      A, #68H            ; 将数据 68H 送入累加器 A
        MOV      SBUF, A            ; 输出数据到 SBUF，启动串行输出
HERE:    SJMP HERE
SISR:    CLR      TI                ; 清零 TI，准备下一次数据发送
        RETI
END

```

在该程序中，在串行中断入口地址处放置一个无条件转向指令，转向中断服务程序，在其中为 TI 清零，准备下一次数据发送。在主程序中，初始化串行口模式 0，开启串行中断，并将数据送到 SBUF，启动串行输出。如果采用 C51 语言进行程序设计，其程序示例如下：

```

#include <reg51.h>        // 头文件

void SISR(void) interrupt 4 // 串行中断服务例程
{
    TI=0;
}

void main(void)           // 主函数
{
    int i;
    i=0x68;                // 初始化数据
    SCON=0x00;             // 初始化串行口模式 0
    ES=1;                  // 开启串行中断
    SBUF=i;                // 将数据送到 SBUF，启动串行输出
    EA=1;                  // 开中断
    while(1)               // 主循环
    {
    }
}

```

在该程序中，定义了串行中断服务程序。在主函数中，初始化串行口模式 0，开启串行中断，并将数据送到 SBUF，启动串行输出。在主循环中不进行任何操作，当数据发送完毕后，将置 TI=1，进入中断服务例程，在其中为 TI 清零，准备下一次数据发送。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

18.6 外部中断源的扩展

51 系列单片机提供了两个外部中断源，对于很多的测控系统，有时外部有很多中断源需要处理，这样外部中断源显然不够用。因此需要进行外部中断源的扩展，从而实现对多个外部中断的响应处理能力。

可以使用定时器/计数器扩展和查询方式扩展两种方法，下面分别进行介绍。

18.6.1 定时器/计数器扩展外部中断源

定时器/计数器扩展外部中断源的原理是当定时器/计数器的计数值达到最大值后，再有一个计数信号输入，便将引发溢出中断。

在实际应用中，将定时器/计数器的初值赋为计数最大值，然后再将定时器/计数器的输入端作为外部中断源的输入端，当外部中断产生的时候，将给定时器/计数器一个计数信号，导致中断响应。

由于定时器/计数器有多种工作模式，对于不同的工作模式有不同的处理方法，主要有以下两种处理方式。

1. 定时器/计数器模式 2 扩展

定时器/计数器的工作模式 2 是一种 8 位自动装载模式。计数器的低 8 位为计数部分，高 8 位为自动装载的计数初值。使用该模式作为外部中断源的扩展的流程，如图 18-8 所示。其具体操作步骤如下：

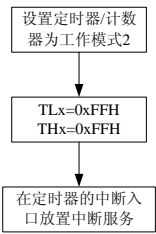


图 18-8 模式 2 的扩展流程

- (1) 首先置 T0 或 T1 为工作模式 2。
- (2) 将该定时器/计数器的初值均赋为 FFH。
- (3) 将外部中断源的中断信号接到定时器/计数器的计数输入端。
- (4) 在相应的定时器/计数器的中断入口处，放置中断服务程序。

下面举例讲解使用定时器/计数器模式 2 扩展外部中断源。使用定时器/计数器 T0 扩展外部中断，由 P3.4 引脚作为外部中断的输入。当 P3.4 引脚有一个负跳变信号时，将 P1 端口反相。

如果采用汇编语言进行程序设计，其程序示例如下：

```
ORG      0000H
AJMP     MAIN
ORG      000BH
AJMP     SISR
ORG      0200H
MAIN:    MOV     TMOD, #06H      ;初始化 T0 为计数功能，模式 2
         MOV     PCON, #80H     ;设置 SMOD=1
         MOV     TL0, #0FFH     ;初值
         MOV     TH0, #0FFH
         SETB    TR0            ;开启计数器 T0
         SETB    ET0            ;开计数器 T0 中断
         SETB    EA            ;开总中断
HERE:    SJMP    HERE
SISR:    MOV     A, P1
         CPL     A              ;反相
```

```
MOV      P1,A
RETI
END
```

在该程序中，T0 中断的入口地址放置跳转指令，指向中断服务例程 SISR。主程序中初始化 T0 为计数功能，模式 2，设置 SMOD=1，并设置计数初值为最大。最后开启计数器 T0 以及相应的中断。当 P3.4 引脚有一个负跳变信号时，T0 立刻溢出，产生中断，在相应的中断服务例程中将 P1 端口反相。如果采用 C51 语言进行程序设计，其程序示例如下：

```
#include <reg51.h>

void T0ISR(void) interrupt 1           //定时器 T0 当做外部中断响应
{
    P1=~P1;                           //反相
}

void main(void)                       //主函数
{
    TMOD=0x06;                        //初始化 T0 为计数功能，模式 2
    PCON|=0x80;                       //设置 SMOD=1
    TL0=0xFF;                         //初值
    TH0=0xFF;                         //初值
    TR0=1;                            //开启计数器 T0
    ET0=1;                            //开计数器 T0 中断
    EA=1;                             //开总中断

    while(1)                          //主循环
    {
    }
}
```

在上面的程序中，自定义了定时器 T0 的中断服务例程，用作扩展的外部中断。在主函数中初始化 T0 为计数功能，模式 2，设置 SMOD=1，并设置计数初值为最大。最后开启计数器 T0 以及相应的中断。程序的主循环不做任何操作。当 P3.4 引脚有一个负跳变信号时，T0 立刻溢出，产生中断，在相应的中断服务例程中将 P1 端口反相。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

2. 其他模式扩展

定时器/计数器的其他工作模式，操作流程和模式 2 类似，不过需要手动重新赋初值。其作为外部中断源的扩展的流程，如图 18-9 所示。

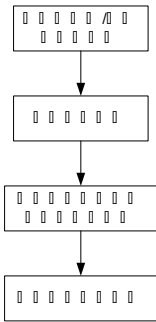


图 18-9 其他模式的扩展流程

下面以定时器/计数器模式 0 为例讲解如何使用该模式扩展外部中断源。使用定时器/计数器 T0 扩展外部中断，由 P3.4 引脚作为外部中断的输入。所实现的功能和上例一样。如果采用汇编语言进行程序设计，其程序示例如下：

```
ORG      0000H
AJMP     START           ;转向主程序
ORG      000BH
AJMP     ITRU            ;转向中断子程序
ORG      0200H
START:   MOV     SP,#60H
        MOV     TMOD,#04H      ;设置计数器 0 模式 0
        MOV     TL0,#1FH      ;初始化
        MOV     TH0,#0FFH
```

```

        SETB    TR0
        SETB    ET0

        SETB    EA                                ;开中断
HERE:    SJMP    HERE
        ORG     0400H

ITRU:    MOV     TL0,#1FH                        ;重置初值
        MOV     TH0,#0FFH
        MOV     A,P1

        CPL     A                                ;反相
        MOV     P1,A
        RETI
END
```

在上面的中断服务子程序中，首先将定时器的最大初值再次装入定时器，以便于开始下一次定时，然后通过取反 P1 端口得到输出。在主程序中，则设置计数器 0 为模式 0，并初始化计数初值为最大，开启动定时器，开总中断。该程序在执行时，如果 P3.4 引脚有一个负跳变信号时，T0 立刻溢出，产生中断，在相应的中断服务例程中将 P1 端口反相。如果采用 C51 语言进行程序设计，其程序示例如下：

```

#include <reg51.h>                                //头文件

void T0ISR(void) interrupt 1                      //定时器 T0 中断响应
{
    TL0=0x1F;                                     //重置计数初值
    TH0=0x0FF;
    P1=~P1;                                       //反相
}

void main(void)                                  //主函数
{
    TMOD=0x04;                                   //设置定时器 T0 为模式 0
    TL0=0x1F;                                    //初始化
    TH0=0x0FF;
    TR0=1;
    ET0=1;
    EA=1;                                        //开中断
    while(1)                                     //主循环
    {
    }
}
```

在上面的程序中，自定义了定时器 T0 的中断服务例程，用做扩展的外部中断。在主函数中初始化 T0 为计数功能，模式 0，设置 SMOD=1，并设置计数初值为最大。最后开启计数器 T0 及相应的中断。程序的主循环不做任何操作。当 P3.4 引脚有一个负跳变信号时，T0 立刻溢出，产生中断，在相应的中断服务例程中将 P1 端口反相。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

使用定时器/计数器扩展外部中断源的优点是，不需要额外的硬件，只使用单片机本身的资源即可完成，程序处理也比较简单；缺点是这种方式占用了本身的定时器/计数器资源，如果需要使用定时器/计数器或更多外部中断源的时候便无法有效使用。

18.6.2 查询方式扩展外部中断源

查询方式扩展外部中断源的示意图，如图 18-10 所示。使用查询方式扩展外部中断源，可以获得较多的外部中断源输入。

图 18-10 中通过 $\overline{\text{INT0}}$ 来扩展了 8 个外部中断源 $\text{INT10} \sim \text{INT17}$ 。这 8 个中断的有效信号为高电平，通过 OC 非门连接到 $\overline{\text{INT0}}$ 端口，外部中断信号同时也分别接到单片机的 P1 端口。

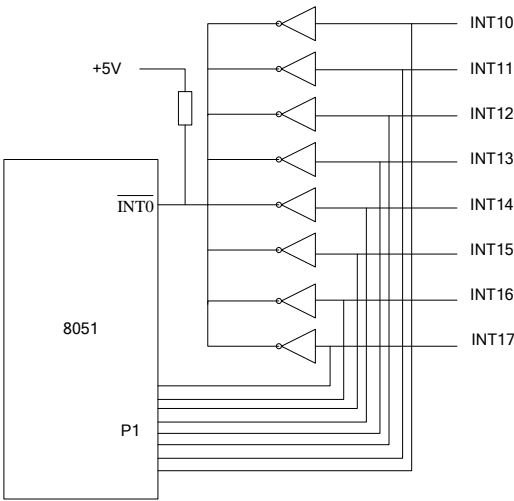


图 18-10 查询方式扩展外部中断源

当某个外部扩展中断发生的时候，输出高电平的有效信号，经反相后转换成为低电平的有效信号输入 $\overline{\text{INT0}}$ 端口。此时 $\overline{\text{INT0}}$ 端口出现负跳变，引发中断。此时，CPU 响应中断，并开始扫描 P1 端口的电平，以确定哪一个中断提出申请。

CPU 对端口的扫描顺序不同，将导致扩展的外部中断的优先级不同。如果端口 P1 的扫描顺序为 P1.0~P1.7，则中断的优先级按照扫描端口的顺序排列。即先扫描的端口优先级高。

下面以 C51 语言为例讲解如何在程序中识别扩展外部中断。程序示例如下：

```
#include<reg51.h>                                     //头文件

void ISR0(void) interrupt 0                             //外部中断 0 服务例程
{
    switch(P1)                                           //扫描端口，判断外部扩展中断
    {
        case 0x01:                                     //INT10 中断
            break;
        case 0x02:                                     //INT11 中断
            break;
        case 0x04:                                     //INT12 中断
            break;
        case 0x08:                                     //INT13 中断
            break;
        case 0x10:                                     //INT14 中断
            break;
        case 0x20:                                     //INT15 中断
            break;
        case 0x40:                                     //INT16 中断
            break;
        case 0x80:                                     //INT17 中断
            break;
        default:
            break;
    }
}
```

```
void main(void)                                //主函数
{
    IP=0x05;                                    //外部中断 0 设置为高优先级
    IT0=1;                                       //外部中断 0 为下降沿触发
    EX0=1;                                       //开 EX0 中断
    EA=1;
    while(1)                                    //主循环
    {
    }
}
```

在该程序中，外部中断 0 设置为高优先级，下降沿触发，并打开相应的中断；而在中断服务例程中使用 switch 语句判断具体是哪个扩展中断发出请求，并分别进行程序处理。

上面的程序可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

使用查询方式扩展外部中断源，可以实现多个外部中断源的扩展，但是需要外接其他硬件，系统和程序都比较复杂。

18.7 小结

本章详细讲述了中断系统的基本概念，并重点介绍了 51 系列单片机的中断类型及中断的各种控制标准位；接着讲述了 51 系列单片机对中断的处理过程，并通过实例详细讲述了各种中断源的编程方式；最后还介绍了外部中断源的扩展方式及其程序设计。中断是 51 系列单片机重要的系统资源，合理使用中断系统，可以减轻 CPU 的负担，简化程序设计，实现对外部信号的实时处理。因此，熟练掌握本章内容是学习 51 系列单片机的基础。

第19章 51 系列单片机的串行接口

串行接口是单片机与外部设备之间进行数据通信的主要途径。51 系列单片机提供了功能强大的全双工串行通信接口，可以方便地实现多机通信或单片机与主机之间的通信。单片机的串行通信接口简单，需要的传输线少，特别适用于远程通信和分布式控制系统中，是单片机之间通信的主要方式。本章将主要介绍 51 系列单片机的串行通信的方式、串行端口的结构、控制寄存器及波特率的设置。本章还将详细介绍串行端口的 4 种工作模式及其程序设计。

19.1 串行通信概述

单片机和外部设备可以采用并行通信和串行通信两种方法进行数据传输。这两种数据传输方式各有其优缺点。

- 并行通信是指数据的各个二进制位同时进行传输。并行通信的示意图，如图 19-1 所示。这种通信方式的优点是传输速度快，效率高，缺点是需要比较多的数据线，数据有多少位就需要多少根数据线，另外并行的数据线易受外界干扰，传输距离不能太远。
- 串行通信是指数据的各个二进制位按照顺序一位一位地进行传输。串行通信的示意图，如图 19-2 所示。这种通信方式的优点是所需的数据线少，节省硬件成本及单片机的引脚资源，并且抗干扰能力强，适合于远距离数据传输，其缺点是每次发送一个比特，导致传输速度慢，效率低。

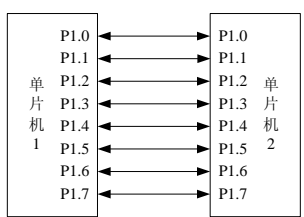


图 19-1 并行通信

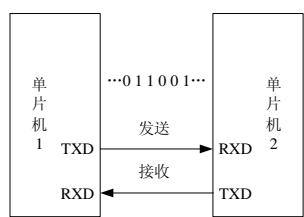


图 19-2 串行通信

并行通信和串行通信的概念广泛应用于现代电子设计中，是最基本的通信方式。两个单片机之间、单片机和计算机之间，以及两台计算机之间都可以采用并行接口和串行接口进行通信。

19.1.1 串行通信简介

单片机的串行通信是将数据的二进制位，按照一定的顺序进行逐位发送，接收方则按照对应的顺序逐位接收，并将数据恢复出来。单片机的串行通信有异步通信和同步通信两种基本方式。下面分别进行介绍。

1. 异步通信方式

异步通信是一种利用数据或字符的再同步技术的通信方式，其全称为 **Asynchronous Communication**。在异步通信过程中，数据通常是以帧为单位进行传送的，每个帧为一个字符或一个字节。发送方将字符帧一位一位地发送出去，接收方则一位一位地接收该字符帧。发送方和接收方各自有一个控制发送与接收的时钟，这两个时钟不同步，互相独立。

在进行异步串行通信时，字符帧的格式，如图 19-3 所示。

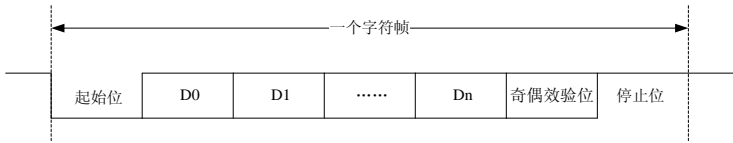


图 19-3 异步通信中字符帧的格式

一个字符帧按顺序一般可以分为 4 部分，即起始位、数据位、奇偶校验位和停止位。下面分别介绍各位的含义。

➤ 起始位

起始位位于字符帧的开始，用于表示向接收端开始发送数据。起始位占用 1 位，为低电平 0 信号。

➤ 数据位

数据位即需要发送的数据。根据需要数据位可以是 5 位、6 位、7 位或 8 位数据，发送时首先发送低位，即低位在前，高位在后。

➤ 奇偶校验位

奇偶校验位为可编程位，用来表明串行数据是采用奇校验还是偶校验。在字符帧中，奇偶校验位只占 1 位。

➤ 停止位

停止位位于字符帧的末尾，表示一帧信息的结束。停止位可以取 1 位、1 位半或 2 位，其为高电平 1 信号。

由于异步串行通信的双方没有同步的时钟，因此在单片机进行异步通信的之前，需要通信的双方统一通信格式。通信格式主要表现在字符帧的格式和通信波特率两个方面。下面分别介绍。

➤ 字符帧（Character Frame）

字符帧格式是字符的编码形式、奇偶校验形式及所采用的起始位和停止位的定义。例如，在传送 ASCII 码数据时，起始位占 1 位，有效数据位取 7 位，奇偶校验位占 1 位，停止位取 1 位。这样一个字符帧共 10 位。通信的双方必须采用相同的字符帧格式。

➤ 波特率（Baud Rate）

波特率指的是每秒发送的二进制位数，单位为 bit/s，即位/秒，常称为波特。波特率是串行通信的重要指标，表明了数据传输的速度。波特率越高，数据传输速度也就越快。

这里需要注意的是，波特率和字符的实际传输速度不相同，波特率等于一个字符帧的二进制编码的位数乘以字符/秒。例如，对于上面的 ASCII 码，一个字符帧用 10 位编码，如果波特率为 1200bit/s，则实际的字符传输速度为 $1200/10=120$ 字符/秒。通信的双方必须采用相同的波特率。

在异步通信的过程中，数据在传输线路上的传送一般是不连续的，即传输时，字符间隔不固定，各个字符帧可以是连续发送，也可以是间断发送。在间断发送时，停止位之后，传输线路上自动保持高电平。

异步串行通信的优点是不需要进行时钟同步，字符帧的长度不受限制，使用起来比较简单，应用范围广；其缺点是传送每个字符都要有起始位、奇偶校验位和停止位，这样便降低了有效的数据传输速率。

2. 同步通信方式

同步通信是一种连续的串行传输数据的通信方式，其全称为 Synchronous Communication。同步串行通信的一次通信过程只传送一帧的信息。这里的帧和异步串行通信的帧具有不同的含义。

同步通信由同步字符、数据字符和校验字符三部分组成。同步通信是把要发送的数据按顺序连接成一个数据块，在数据块的开头附加同步字符，在数据块的末尾附加差错校验字符。在

数据块的内部，数据与数据之间没有间隔。

按照同步字符的个数，同步串行通信的字符帧有两种结构，分别为单同步字符帧结构和双同步字符帧结构，如图 19-4 所示。

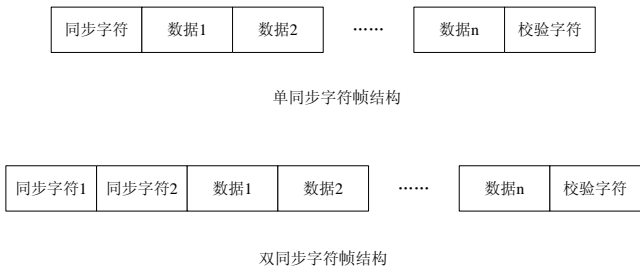


图 19-4 同步通信的字符帧格式

在进行同步串行通信时，发送方首先发送同步字符，数据则紧跟其后发送。接收方检测到同步字符后，开始逐个接收数据，直到所有数据接收完毕，然后按照双方规定的长度恢复成一个一个的数据字节。最后进行校验，如果无传输错误，则可以结束一帧的传输。

在进行同步串行通信时，需要注意如下几点。

- 同步串行通信的过程中，数据块之间一般不能有间隔，如果需要间隔，则应发送同步字符来填充间隔。
- 在同步串行通信中，同步字符应该采用统一的格式。例如，在单同步字符帧结构中，同步字符常采用 ASCII 码中规定的 SYN 代码，即 16H；在双同步字符帧结构中，同步字符一般采用国际通用标准代码，即 EB90H。当然，也可以由通信的双方共同规定同步字符的格式。
- 同步串行通信过程中，发送方和接收方需要采用统一的时钟，以保持完全的同步。一般来说，如果是近距离数据传输，则可以在发送方和接收方之间增加一根公用的时钟信号线来实现同步；如果是远距离数据通信，则可以通过解调器从数据流中提取出同步信号，采用锁相技术使接收方获得和发送方完全相同的时钟信号，从而实现同步。

同步串行通信的优点是不用单独发送每个字符，其传输速率高，一般用于高速率的数据通信场合；缺点是需要进行发送方和接收方之间的时钟同步，整个系统设计比较复杂。

19.1.2 串行通信的数据传送方式

根据通信双方之间的数据流向，串行通信可以分为三种制式，即单工制式、半双工制式和全双工制式。这三种制式的示意图，如图 19-5 所示。下面分别进行介绍。

➤ 单工制式（Simplex）

单工制式是单向的数据传送方式。在通信的双方中，一方固定为发送端，另一方固定为接收端。单工方式比较简单，只需要 1 根数据线便可以工作进行。

➤ 半双工制式（Half Duplex）

半双工制式一种准双向的数据传送方式。在半双工制式中，系统通信的双方都有一个串行发送器和串行接收器，通过内部的切换开关实现收方的转换。半双工的数据传输是双向的，数据即可以从 A 端发送到 B 端，也可以从 B 端发送到 A 端。但在同一时间内只能从一端发送到另一端。半双工串行数据传输方式只需要 1 根数据线，便可以实现分时双向数据收发。

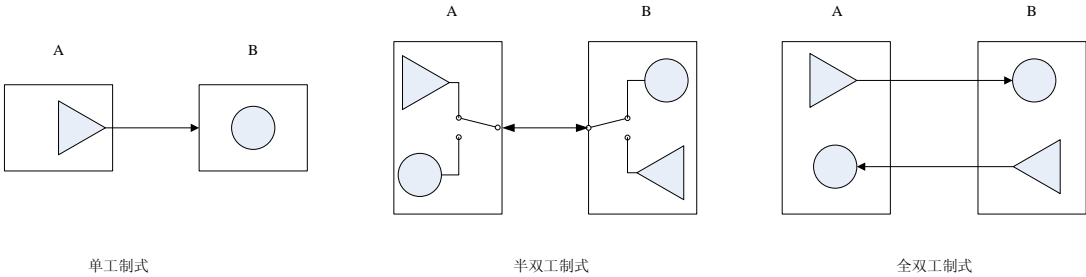


图 19-5 串行通信的制式

➤ 全双工制式（Full Duplex）

全双工制式是一种双向的数据传送方式。在全双工制式中，系统通信的双方都有一个串行发送器和串行接收器。与半双工制式不同的是，这里通信的双方之间采用两根数据线，分别用于 A 发送 B 接收和 B 发送 A 接收，这样便可以实现同时的双向数据传输。

51 系列单片机内部集成有全双工串行通信端口，可以实现多机全双工数据通信。但在一些简单的应用场合中，大都采用半双工制式，这样虽然没有充分发挥硬件的功能，但系统的程序设计比较简单。

19.2 51 系列单片机的串行接口

51 系列单片机内部集成的全双工串行通信接口电路，常称为 UART。该串行接口电路功能很强，不仅可以进行串行异步数据的发送和接收，也可以作为一个同步移位寄存器使用。

19.2.1 单片机串行接口的内部结构

51 系列单片机的全双工串行口主要由数据发送缓冲器、发送控制器 TI、输出控制门、接收控制器、输入移位寄存器、数据接收缓冲器等组成，如图 19-6 所示。

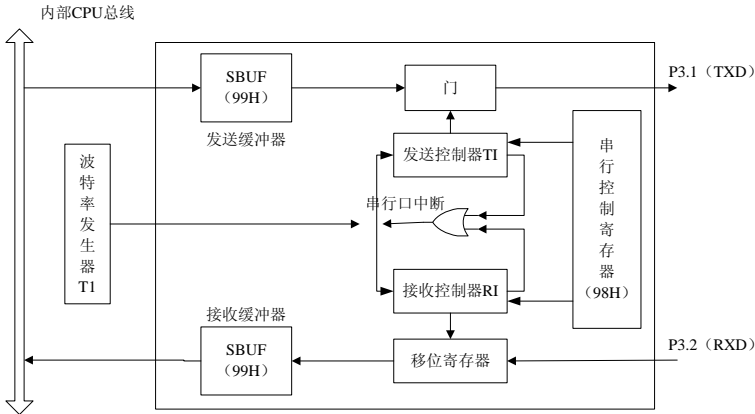


图 19-6 单片机串行口内部结构

串行接口内部包含有两个互相独立的发送和接收缓冲器，可以在同一时刻进行数据的发送和接收。发送缓冲器只能写入而不能读出数据，而接收缓冲器则只能读出而不能写入数据。因此，在单片机的指令系统中，两个缓冲器可以共用一个符号 SBUF，占用同一个地址 99H。一般通过不同的读缓冲器和写缓冲器指令来决定是对哪一个缓冲器进行操作。读缓冲器和写缓冲器指令分别如下：

```
MOV      SBUF,A           ;写缓冲器
```

MOV A, SBUF ;读缓冲器

其中，第一个指令对发送缓冲器进行操作，将累加器 A 中的数据写入发送缓冲器。第二个指令对接收缓冲器进行操作，将读取接收缓冲器中的数据并送入累加器 A。

单片机的串行接口主要完成串行数据收发工作。在串行数据发送的时候，由 CPU 执行上面的写指令把数据写入发送缓冲器，则串行接口电路便自动启动，从而一位一位地向外传送数据。在发送的同时，接收端也可以一位一位地接收数据，直到把一组串行数据接收完毕，送入缓冲器，然后自动通知 CPU，CPU 执行上面的读指令便可以把接收缓冲器中的数据读入。

在整个串行数据的发送和接收过程中，CPU 的占用时间比较少，便于进行其他操作。

51 系列单片机是通过特殊功能寄存器的设置、检测和读取来管理串行通信接口的。单片机的串行接口有两个特殊功能寄存器 **SCON** 和 **PCON**。**SCON** 用于存放串行接口的控制和状态信息，**PCON** 用于改变串行接口的波特率。51 系列单片机的波特率发生器可以由定时器 **T1** 或 **T2** 来构成。

19.2.2 单片机串行接口的程序控制

51 系列单片机的串行通信接口，通过控制寄存器 **SCON** 和波特率选择特殊功能寄存器 **PCON** 来控制。下面分别介绍这两个寄存器。

1. 串行接口控制寄存器 SCON

串行控制寄存器 **SCON** 用于选择串行通信的工作方式和某些控制功能, 包括接收/发送控制及设置状态标志等。寄存器 **SCON** 的格式及各位的含义, 如图 19-7 所示。

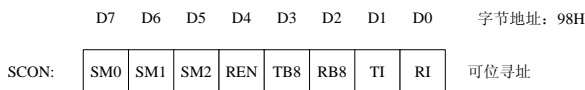


图 19-7 SCON 定义

控制寄存器 SCON 的字节地址为 98H，可进行位寻址。下面分别介绍每一位的用途。

- **SM0 和 SM1**: 串行通信接口工作方式选择位。**SM0** 和 **SM1** 共有 4 种组合, 分别对应 4 种工作方式, 如表 19-1 所示。

表 19-1 SM0 与 SM1 的组合工作方式

SM0	SM1	工作方式	功能说明	波特率
0	0	模式 0	同步移位寄存器方式	$f_{osc}/12$
0	1	模式 1	10 位异步发送接收方式	可变, 由定时器控制
1	0	模式 2	11 位异步发送接收方式	$f_{osc}/32$ 或 $f_{osc}/64$
1	1	模式 3	11 位异步发送接收方式	可变, 由定时器控制

其中 f_{osc} 为单片机系统的主振频率。

- **SM2**: 多机通信的控制位。在多机通信模式中, 通信的双方分为主机和从机, 而 **SM2** 用于控制从机的接收。当 **SM2=1** 时, 允许多机串行通信, 主要用于模式 2 和模式 3 中。

当 SM2 设置为 1 时，则从机可以接收地址帧，若接收到的第 9 位数据（RB8）为 0 时，表示是数据帧，不启动中断标志 RI，即 RI=0，并且将前面接收到的 8 位数据丢弃；如果接收到的第 9 位数据（RB8）为 1 时，表示是地址帧，此时将接收到的前 8 位数据送入 SBUF 中，并置位中断标志 RI=1，进行中断申请。

当 SM2 设置为 0 时，从机可以接收所有的信息。从机在接收到一帧的数据后，无论第 9 位数据（RB8）是 0 还是 1，都将启动中断标志 RI，即 RI=1，将接收到的数据送入 SBUF 中。

在模式 0 中, SM2 必须设置为 0。而在模式 1 中, 如果 SM2=1, 则只有接收到有效停止位时, 才启动中断标志 RI, 即置 RI=1, 以便于接收下一帧数据。

- **REN**: 接收允许/禁止控制位, 相当于串行数据接收的开关。**REN** 可以由软件进行置 1 或清零, 当 **REN**=1 时, 允许接收数据; 当 **REN**=0 时, 禁止接收数据。

➤ **TB8**: 主要用于在模式 2 和模式 3 中, 作为发送数据的第 9 位, 可以根据发送数据的需要由软件置位或复位。在模式 0 和模式 1 中, 该位不使用。

在单机通信中, **TB8** 可以作为奇偶校验位。而在多机串行通信中, 可以作为发送地址帧和数据帧的标志位。一般规定发送地址帧的时候, 设置 **TB8=1**; 发送数据帧的时候, 设置 **TB8=0**。

➤ **RB8**: 主要用于在模式 2 和模式 3 中, 作为接收数据的第 9 位。**RB8** 即可以作为约定的奇偶校验位, 也可以是约定的地址/数据标志位。

在软件中, 可以根据 **RB8** 被置位的情况对接收到的数据进行判断处理。例如在多机通信中, 若 **RB8** 被置 1, 即 **RB8=1**, 则说明收到的数据为地址帧; 如果 **RB8=0**, 则说明收到的数据为数据帧。

在模式 1 中, 如果 **SM2=0**, 即采用的不是多机通信模式, 则 **RB8** 是已接收到的停止位。在模式 0 中, 该位不使用。

➤ **TI**: 发送中断请求标志位, 在一帧数据信息发送结束的时候由硬件自动置位。**TI=1** 表示发送缓冲器已空, 通知 CPU 可以发送下一帧数据。**TI** 可以作为 CPU 查询, 以决定 CPU 是否需要从 **SBUF** 中提取接收到的数据, 也可以用做中断申请标志位。

在不同的模式下, **TI** 置位的情况不同。在模式 0 中, 串行发送完 8 位数据时便进行硬件置位; 其他模式中, 串行发送到停止位的时候由硬件置位。**TI** 不会自动复位, 需要用软件复位清零。

➤ **RI**: 接收中断请求标志位, 在接收到一帧的有效数据后由硬件置位。**RI=1** 表示接收缓冲器已满, 即一帧数据接收完毕, 并已装入接收缓冲器中, 通知 CPU 可以将数据取走。**RI** 可以作为查询, 也可以作为中断申请标志位。

在不同的模式下, **RI** 置位的情况不同。在模式 0 中, 串行接收完 8 位数据时便进行硬件置位; 其他模式中, 串行接收到停止位的时候由硬件置位。同样, **RI** 不会自动复位, 需要在软件中手动复位清零, 以便于接收下一帧数据。

由于 **SCON** 可以位寻址, 因此在汇编语言程序中, 可以采用如下语句设置 **TCON** 寄存器。

```
SETB SM0
SETB SM1
```

该语句采用位寻址的方式, 分别置位 **SM0** 和 **SM1**, 用于设置串口为工作模式 3。

在 C51 语言中, 头文件 **REG51.H** 包含了 **TCON** 的位定义, 示例如下:

```
sfr      SCON  = 0x98;           //定义 SCON
sbit     SM0   = SCON^7;         //串行通信接口工作方式选择位
sbit     SM1   = SCON^6;         //串行通信接口工作方式选择位
sbit     SM2   = SCON^5;         //多机通信的控制位
sbit     REN   = SCON^4;         //接收允许/禁止控制位
sbit     TB8   = SCON^3;         //发送数据的第 9 位
sbit     RB8   = SCON^2;         //接收数据的第 9 位
sbit     TI    = SCON^1;         //发送中断请求标志位
sbit     RI    = SCON^0;         //接收中断请求标志位
```

因此, 在 C51 语言可以直接对这些位进行操作, 示例如下:

```
SM0=1
SM1=1
```

该语句直接置位 **SM0** 和 **SM1**, 用于设置串口为工作模式 3。

也可以直接为寄存器 **SCON** 赋值来设置串口, 示例如下:

```
SCON=0x50;           //串口模式 1, 允许接收
```

该语句设置串口为工作模式 1, 允许接收。

2. 特殊功能寄存器 PCON

特殊功能寄存器 PCON 用于 CHMOS 型单片机中进行电源控制，也称为电源控制寄存器。PCON 的格式，如图 19-8 所示。PCON 的单元字节地址为 87H，不可以进行位寻址。

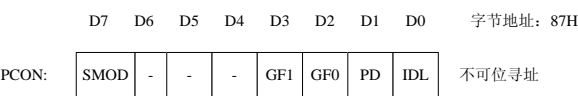


图 19-8 PCON 的格式

其中，D0~D3 表示 CHMOS 型单片机的掉电控制位，D4~D6 不使用，D7（SMOD）为波特率倍增位。在模式 1、模式 2 和模式 3 下，如果置 SMOD=1，则波特率提高一倍；如果 SMOD=0，则波特率不变。

对于 HMOS 的 51 系列单片机，PCON 中只有 SMOD 起作用，其他位都不使用。在汇编语言中，可以使用 MOV 指令来对寄存器 PCON 进行操作，示例如下：

```
MOV      PCON, #80H                //设置 SMOD=1
```

该语句为 PCON 赋值 0x80，即置 SMOD=1，波特率加倍。在 C51 语言中，可以直接对 PCON 进行赋值，示例如下：

```
PCON|=0x80;                        //设置 SMOD=1
```

该语句直接为 PCON 赋值 0x80，即置 SMOD=1，波特率加倍。

19.2.3 波特率的程序设计

波特率是异步串行通信的重要概念。在单片机的串行通信中，通信的双方需要有共同的波特率。

51 系列单片机的串行口共有 4 种工作模式，对应有三种波特率。其中，模式 0 和模式 2 具有固定的波特率；而模式 1 和模式 3 的波特率是可变的，一般由定时器 T1 或 T2 的溢出率来决定。下面分别介绍各种模式下的波特率的计算及程序设计。

1. 模式 0 的波特率

串口的工作模式 0 是同步移位寄存器方式。在模式 0 下，单片机的每个机器周期产生一个移位时钟，对应着一位数据的发送和接收。因此，这种模式下，波特率固定为单片机振荡频率的 1/12，其波特率的计算公式如下：

模式 0 波特率= $f_{osc}/12$

在模式 0 下，波特率不受波特率倍增位 SMOD 的影响。由于波特率固定，因此，在程序设计时只要指定串口工作于模式 0 即可，而无须在程序中设置波特率。

例如，对于 12MHz 的外部晶体振荡频率，模式 0 可以获得 1Mbit/s 的波特率。

2. 模式 2 的波特率

串口的工作模式 2 是 11 位异步发送接收方式。在模式 2 下，波特率由单片机的振荡频率 f_{osc} 和 PCON 的波特率倍增位 SMOD 共同决定。模式 2 下的波特率的计算公式如下：

模式 2 波特率= $f_{osc} \cdot 2^{SMOD} / 64$

从公式中可以看出，当 SMOD=0 时，波特率为 $f_{osc}/64$ ，当 SMOD=1 时，波特率为 $f_{osc}/32$ 。

例如，对于 12MHz 的外部晶体振荡频率，通过寄存器 PCON 可以选择波特率。如果采用汇编语言，示例如下：

```
MOV      PCON, #00H                ;设置 SMOD 为 0
MOV      PCON, #80H                ;设置 SMOD 为 1
```

其中，第一条指令设置 SMOD=0，可以获得 187.5kbit/s 的波特率。而第二条指令设置

51 单片机开发与应用技术详解

SMOD=1，可以获得 375kbit/s 的波特率。

在 C51 语言中，则可以采用如下的赋值语句来达到同样的效果。

```
PCON=0x00;           //设置 SMOD=0
PCON=0x80;           //设置 SMOD=1
```

3. 模式 1 和模式 3 的波特率

串口的工作模式 1 和工作模式 3 分别为 10 位异步发送接收方式和 11 位异步发送接收方式。其串行移位时钟脉冲由定时器 T1 的溢出率来决定，因此，波特率由定时器 T1 的溢出率和 SMOD 来共同决定。模式 1 和模式 3 的波特率计算公式如下：

模式 1 (或模式 3) 波特率 = T1 溢出率 $2^{SMOD}/32$

模式 1 和模式 3 的波特率需要对定时器 T1 进行工作方式设置，以便于得到需要的波特率发生器。最常用的是，使 T1 工作于模式 2，这是初值自动加载的定时方式。如果计数器的初始值为 X，则每过 256-X 个机器周期的时候，定时器 T1 便将产生一次溢出，溢出的周期为 $(256-X) \times 12/f_{osc}$ 。

单片机的溢出率是溢出周期的倒数，即 $\frac{f_{osc}}{12(256-X)}$ ；

由前面的波特率计算公式，便有 波特率 = $\frac{2^{SMOD}}{32} \times \frac{f_{osc}}{12(256-X)}$ 。

从上面的式子可以解出，T1 工作在模式 2 下的初值 X 的计算公式如下：

$$X = 256 - \frac{f_{osc}(SMOD+1)}{384 \times \text{波特率}}$$

下面举一个例子，对于 8051 单片机（或者其他兼任型号的 51 系列单片机），外接 11.0592MHz 的晶振，采用内部振荡器工作模式。这里使用工作于模式 2 的定时器 T1，作为串行通信的波特率发生器，波特率为 2400bit/s。

如果不使用波特率倍增位 SMOD，则设置 SMOD=0，则根据上面的公式，可知

$$X = 256 - \frac{11.0592 \times 10^6 \times (0+1)}{384 \times 2400} = 244 = F4H$$

因此，可以设置 (TH1) = (TL1) = F4H。

如果采用汇编语言进行程序设计，这个例子中波特率的初始化程序如下：

```
MOV    TMOD,#20H           ;设置定时/计数器 1 定时，工作于方式 2
MOV    TH1,#F4H             ;设置定时/计数器 1 的初始值
MOV    TL1,#F4H
SETB   TR1                  ;启动定时/计数器 1 开始定时计数
MOV    PCON,#00H           ;设置 SMOD 为 0
MOV    SCON,#50H           ;设置串行工作方式 1，允许接收
```

上面的程序中，首先设置定时/计数器 T1 工作于方式 2，并设置初值，接着启动 T1。最后设置串口工作方式 1，并且允许接收。

如果采用 C51 语言进行程序设计，这个例子中波特率的初始化程序如下：

```
TMOD=0x20;           //设置定时/计数器 1 定时，工作于方式 2
TH1=0xF4;             //设置定时/计数器 1 的初始值
TL1=0xF4;
TR1=1;                //启动定时/计数器 1 开始定时计数
PCON=0x00;           //设置 SMOD 为 0
```



```
SCON=0x50; //设置串行工作方式 1，允许接收
```

如表 19-2 所示列出了模式 1 和模式 3 下，采用定时器 T1 的工作模式 2 作为波特率发生器时，一些常用波特率的参数及初值设置。

表 19-2 常用波特率参数设置

波特率 (bit/s)	f _{osc} (MHz)	SMOD	定时器 T1 工作模式	初 值
110	6	0	2	72H
110	12	0	2	FEEDH
137.5	11.986	0	2	1DH
1200	11.0592	0	2	E8H
2400	11.0592	0	2	F4H
4800	11.0592	0	2	FAH
9600	11.0592	0	2	FDH
19200	11.0592	1	2	FDH
62500	12	1	2	FFH

其中，很多都是用了 11.0592MHz 的晶体振荡频率，这是因为这个频率可以使定时器 T1 的初值设置为整数，便于产生精确的波特率。

为了获得很低的串行通信波特率，可以采用定时器 T1 的模式 0 和模式 1，即采用 13 位或 16 位定时方式。在这种情况下，T1 溢出后，需要重新装入初值，从而多加了语句的执行，消耗了 CPU 的时间，因此波特率将会产生一定的误差。因此，一般不推荐使用这种工作方式。

另外，对于 52 子系列的单片机，除了选择定时器/计数器 T1 外，还可以选择 T2 作为串行口的波特率发生器，通过设置 T2CON 寄存器中的 TCLK 位和 RCLK 位即可。

- 当 TCLK=1 时，T2 被选择用于发送波特率发生器；
- 当 RCLK=1 时，T2 被选择用于接收波特率发生器；
- 当 TCLK 和 RCLK 同时都置 1 时，T2 作为发送/接收波特率发生器。

用 T2 作波特率发生器的时候，波特率与波特率倍增位 SMOD 无关，只与 T2 的溢出率有关。T2 的计数时钟可以是单片机的内部，也可以从单片机外部输入，取决于 T2CON 寄存器中的 C/T2 位的值，并且 T2 的溢出信号要经过一个 16 分频的计数器。下面分别介绍不同时钟信号的波特率计算公式。

- 当 C/T2=0 时，选择单片机内部时钟作为 T2 的计数时钟，计数频率为 f_{osc}/2，波特率的计算公式如下：

波特率 =
$$\frac{f_{osc}}{2 \times 16 \times [65536 - (RCAP2H, PCAP2L)]}$$

式中，RCAP2H 和 RCAP2L 为 T2 的自动重装载值。

- 当 C/T2=1 时，选择单片机外部时钟作为 T2 的计数时钟，计数频率为外部时钟的频率，最高为 f_{osc}/24，波特率的计算公式如下：

波特率 =
$$\frac{\text{外部时钟频率}}{2 \times 16 \times [65536 - (RCAP2H, PCAP2L)]}$$

式中，RCAP2H 和 RCAP2L 为 T2 的自动重装载值。

在程序设计中，如果使用汇编语言，则可以按照如下的指令来设置 T2。程序示例如下：

```
MOV TH2, #0BH ; T2 定时器赋预装载值
MOV TL2, #0DCH
MOV RCAP2H, #0BH ; T2 定时器赋预装载值
MOV RCAP2L, #0DCH
SETB TR2 ; 启动定时/计数器 2
```

如果采用 C51 语言进行程序设计，可以直接为寄存器赋值，程序示例如下：

```
TH2=0x0B;           // T2 定时器赋预装载值
TL2=0xDC;
RCAP2H=0x0B;        //重载值
RCAP2L=0xDC;
TR2=1;              //启动 T2 定时器
```

19.3 串行口的工作模式 0

串行口的工作模式 0 是同步移位寄存器输入/输出方式，可以设置控制寄存器 SCON 的 SM0=0 和 SM1=0 来实现。在模式 0 下，波特率固定为 $f_{osc}/12$ 。模式 0 中的数据以 8 位为一帧，没有起始位和停止位，发送时低位在前，高位在后。模式 0 的数据帧格式，如图 19-9 所示。

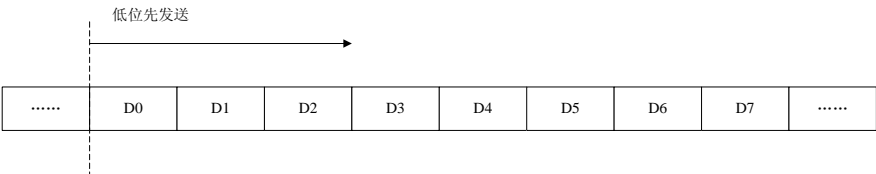


图 19-9 模式 0 的帧格式

19.3.1 模式 0 的发送及扩展输出端口

对于模式 0 的数据发送，TXD 引脚都用于发送同步移位脉冲，而 8 位串行数据是通过 RXD 引脚来输出。此时，常用于扩展单片机的并行 I/O 输出端口。

1. 模式 0 的发送

在模式 0 下，程序可以按照如下的步骤来进行数据的发送。

- (1) 首先，置串行接口控制寄存器 SCON 的 TI=0，启动串行口发送；
- (2) 执行写发送缓冲器指令，示例如下：

```
MOV     SBUF,A
```

单片机的 CPU 执行完这条指令后，在 TXD 端发送同步移位脉冲，在每个机器周期的 S6P2 状态，8 位数据便从 RXD 端由低位到高位逐个发送出去。当 8 位数据发送完毕的时候，硬件自动置中断标志 TI=1，请求中断，表示发送缓冲器已空。TI 不会自动清零，当要发送下一组数据时，必须在软件中置 TI=0，然后才能发送下一组数据。

2. 扩展并行输出口

扩展并行输出口的电路图，如图 19-10 所示。单片机的串行口在模式 0 下，数据以串行方式逐位发出，如果外接一个串入并出的移位寄存器，例如 CD4094，将串行数据恢复为并行数据，便实现了并行输出，即扩展了单片机的并行输出口。

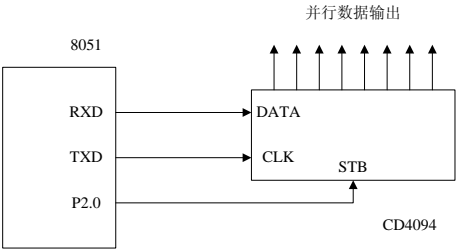


图 19-10 扩展并行输出口

其中，移位寄存器 CD4094 的 STB 为并行数据输出允许控制端，当 STB=1 时，打开输出控制门，实现数据的并行输出。这样可以避免串行输入的时候，并行输出端数据的不稳定。

在程序中，使用串行口进行并行输出之前，需要对寄存器 SCON 进行初始化，即工作模式的设置。由于这里使用的是串行口的模式 0，因此，只需将 00H 送入 SCON 即可。

串行口模式 0 的数据输出可以采用查询方式，也可以采用中断方式。分别介绍如下：

- 在查询方式中，查询 TI 的值，如果 TI=1，则结束查询，可以发送下一组数据；如果 TI=0，则继续查询。
- 在中断方式中，在 TI 置位后产生中断申请，在中断服务程序中发送下一组数据。

可以看出，采用查询方式和中断方式，都需要借助 TI 标志。如果采用汇编语言进行程序设计，采用查询等待的方式，其程序示例如下：

```

                ORG      0100H
MAIN:          MOV      SCON,#00H          ;初始化串行口模式 0
                CLR      ES                ;禁止串行中断
                MOV      A,#67H            ;将数据 67H 送入累加器 A
                CLR      P2.0              ;关闭 CD4094 的并行输出
                MOV      SBUF,A            ;输出数据到 SBUF，启动串行输出
                JNB      TI,$               ;查询 TI，若 TI=0，表示未发送完毕，等待
                SETB     P2.0              ;TI=1，表示发送完毕，启动 CD4094 的并行输出
                CLR      TI                ;清零 TI，准备下一次数据发送
                END
```

在该程序中，首先初始化串行口模式 0，并禁用串行中断，接着将数据发送到 SBUF 启动串行输出。当查询 TI=1 时，表示发送完毕，启动 CD4094 的并行输出。最后 TI 清零，准备下一次数据发送。如果采用 C51 语言进行程序设计，其程序示例如下：

```

#include <reg51.h>                                //头文件

sbit STB=P2^0;                                     //声明 STB

voidmain(void)                                     //主函数
{
    int i;                                         //声明变量
    i=67;                                         //初始化
    SCON=0x00;                                    //初始化串行口模式 0
    ES=0;                                         //禁止串行中断
    STB=0;                                        //关闭 CD4094 的并行输出
    SBUF=i;                                       //输出数据到 SBUF，启动串行输出
    while (TI)                                   //等待 TI=1
    {
        STB=1;                                   //启动 CD4094 的并行输出
        TI=0;                                    //TI 清零
    }
}
```

在该程序中，首先定义 STB 用于控制 CD4094。在主程序中，初始化串行口模式 0，并禁用串行中断，接着将数据发送到 SBUF 启动串行输出。程序中使用 while 语句查询 TI，当 TI=1 时，表示发送完毕，启动 CD4094 的并行输出。最后 TI 清零，准备下一次数据发送。

19.3.2 模式 0 的接收及扩展输入端口

对于模式 0 的数据发送，TXD 引脚都用于发送同步移位脉冲，而 8 位串行数据是通过 RXD 引脚来输入。此时，常用于扩展单片机的并行 I/O 输入端口。

1. 模式 0 的接收

在模式 0 下，程序可以按照如下的步骤来进行数据的接收。

(1) 首先置串行接口控制寄存器 SCON 的 RI=0, REN=1, 启动串行口接收。此时，在 TXD 端发送同步移位脉冲，在同步脉冲为低电平的时候，在每个机器周期的 S6P2 状态，8 位数据从 RXD 端由低位到高位逐位接收。

(2) 当 8 位数据接收完毕的时候，硬件自动置 RI=1, 请求中断，表示接收数据已装入接收缓冲器，可以由 CPU 用指令读取。示例如下：

```
MOV      A, SBUF
```

这里需要注意的是，RI 不会自动清零，当需要接收下一组数据的时候，必须在软件中置 RI=0, 然后才可以接收下一组数据。

另外，在模式 0 工作时，寄存器 SCON 中的 SM2 必须置 0, 而 RB8 位和 TB8 位都不起作用，一般置 0 即可。

2. 扩展并行输入口

利用单片机串口扩展并行输入口的电路图，如图 19-11 所示。其中，在单片机串行口外接一个并入/串出的移位寄存器，如 CD4014, 则可以实现并行数据通过串行口输入，即扩展了并行输入口，此时单片机串行口仍工作于模式 0。

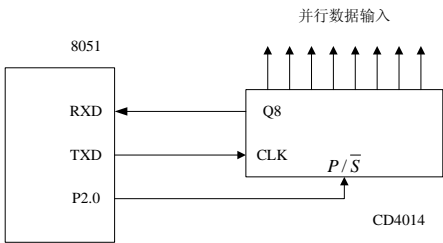


图 19-11 扩展并行输入口

其中，串出移位寄存器 CD4014 的 P/ \bar{S} 为预置/移位控制端，当 P/ \bar{S} =1 的时候，8 位数据并行置入移位寄存器；当 P/ \bar{S} =0 的时候，移位寄存器中的数据串行移位输出。

在程序中，使用串行口进行并行输入之前，同样需要对寄存器 SCON 进行初始化，即工作模式的设置。由于这里使用的是串行口的模式 0 接收，因此，需将 10H 送入 SCON，即置 REN=1。

串行口模式 0 的数据输出可以采用查询和中断两种方式。分别介绍如下：

- 在查询方式中，查询 RI 的值，如果 RI=1 则结束查询，可以接收下一组数据；如果 RI=0，则继续查询。
 - 在中断方式中，在 RI 置位后产生中断申请，在中断服务程序中接收下一组数据。
- 可以看出，采用查询方式和中断方式，都需要借助 RI 标志。

如果采用汇编语言进行程序设计，采用查询等待的方式，其程序示例如下：

```
ORG      0100H
CLR      ES                                ;禁止串行中断
MOV      SCON, #10H                       ;串行口模式 0 的初始化，RI 清零，并启动接收
SETB    P2.0                             ;并行数据送入 CD4014
CLR      P2.0                             ;CD4014 串行数据输出
LOOP:    JNB    RI, LOOP                   ;查询 RI，若 RI=0，表示未接收完，等待
CLR      RI                               ;接收完毕，清零 RI，准备接收下一个数据
MOV      A, SBUF                          ;读取数据到累加器 A
END
```

该程序中，首先串行口模式 0 的初始化，RI 清零，并启动接收。接着使 CD4014 将并行数

据串行输入到单片机串口。程序中查询 RI，当 RI=1 时，表示接收完毕，清零 RI，准备接收下一个数据，并将读取的数据送到累加器 A。

如果采用 C51 语言进行程序设计，其程序示例如下：

```
#include <reg51.h>                                //头文件

sbit P_S=P2^0;                                     //声明 P_S

void main(void)                                     //主函数
{
    int i;
    SCON=0x10;                                     //串行口模式 0 的初始化，RI 清零，并启动接收
    ES=0;                                           //禁止串行中断
    P_S=1;                                          //并行数据送入 CD4014
    P_S=0;                                          // CD4014 串行数据输出
    while (RI)                                     //查询 RI，等待接收完毕
    {
        RI=0;                                     //RI 清零
        i=SBUF;                                   //读出数据
    }
}
```

该程序中，首先定义 CD4014 控制位 P_S。在主程序中初始化串行口模式 0，RI 清零，并启动接收。接着使 CD4014 将并行数据串行输入到单片机串口。程序中使用 while 循环查询 RI，当 RI=1 时，表示接收完毕，清零 RI，准备接收下一个数据，并将读取的数据送到变量 i 中。

19.4 串行口的工作模式 1

串行口的工作模式 1 是波特率可变的串行异步通信方式，可以设置控制寄存器 SCON 的 SM0=0 和 SM1=1 来实现。其中波特率由定时器 T1 的溢出率及 SMOD 位共同决定。工作模式 1 下数据帧的格式，如图 19-12 所示。其由 10 位组成，按顺序分别为起始位、8 位数据位、停止位。数据在传输时，低位在前，高位在后。

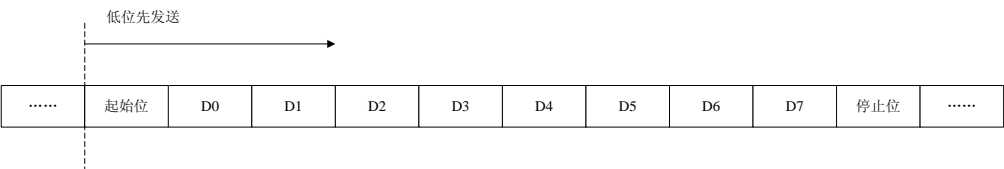


图 19-12 模式 1 的帧格式

19.4.1 模式 1 的发送

在串行口的工作模式 1 中，TXD 引脚为数据发送端。串行异步通信方式的双方不需要时钟同步，发送方和接收方都有自己的移位脉冲。模式 1 发送时的波特率，由定时器 1 的溢出信号和波特率倍增位 SMOD 来共同决定，可以随着定时器初值的不同而变化。

在串行口的工作模式 1 中，数据的发送流程如下：

- (1) 初始化串口及波特率；
- (2) 置串行接口控制寄存器 SCON 的 TI=0，启动串行口发送；
- (3) 执行写发送缓冲器 SBUF 指令，示例如下：

```
MOV     SBUF,A
```

- (4) 硬件自动发送起始位，起始位为逻辑低电平；

51 单片机开发与应用技术详解

- (5) 发送 8 位数据，低位首先发送，高位最后发送；
- (6) 硬件自动发送停止位，停止位为逻辑高电平；
- (7) 程序中清零 TI，以便于发送下一个数据。

在发送移位脉冲的作用下，数据帧依次从 TXD 引脚发出。一帧信息发送完毕后，自动维持 TXD 引脚为高电平。在 8 位串行数据发送完毕后，也就是在插入停止位的时候，使 TI 置 1，用以通知 CPU 可以发送下一帧的数据。如果采用汇编语言，按照上面的步骤，则其程序示例如下：

```

                ORG      0000H
                AJMP     MAIN
MAIN:           MOV      SCON, #50H                ;初始化串行口模式 1
                MOV      TMOD, #20H                ;初始化 T1 为定时功能，模式 2
                MOV      PCON, #80H                ;设置 SMOD=1
                MOV      TL1, #0F4H                ;波特率 4800bit/s，初值
                MOV      TH1, #0F4H
                MOV      IE, #90H
                CLR      ES                        ;禁止串行中断
                SETB     TR1                        ;启动定时器
                MOV      A, #76H
                MOV      SBUF, A
                JNB      TI, $                      ;查询 TI，若 TI=0，表示未发送完毕，等待
                CLR      TI
                MOV      A, #77H
                MOV      SBUF, A
                JNB      TI, $                      ;查询 TI，若 TI=0，表示未发送完毕，等待
                CLR      TI
HERE:           JMP      HERE
                END
```

在该程序中，首先初始化串口为模式 1，然后使用定时器 T1 设置波特率，并且禁止串行中断和启动定时器。接着分别采用 MOV 指令向 SBUF 写数据，数据将通过串口输出。如果采用 C51 语言进行程序设计，按照上面的步骤，则其程序示例如下：

```

#include <reg51.h>                                //头文件

voidmain(void)                                    //主函数
{
    SCON=0x50;                                    //初始化串行口模式 1
    TMOD=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                     //波特率 4800bit/s，初值
    TH1=0xF4;
    ES=0;                                          //禁止串行中断
    TR1=1;                                         //启动定时器
    SBUF=0x76;
    while(TI)                                     //等待 TI=1
    {
        TI=0;                                     //TI 清零
    }
    SBUF=0x77;
    while(TI)                                     //等待 TI=1
    {
        TI=0;                                     //TI 清零
    }
}
```

```
}  
}
```

在该程序中，主函数首先初始化串口为模式 1，然后使用定时器 T1 设置波特率，并且禁止串行中断和启动定时器。接着分别为 SBUF 赋值，数据将通过串口输出。

19.4.2 模式 1 的接收

在串行口的工作模式 1 中，RXD 引脚为数据接收端。模式 1 接收数据中的定时信号可以有两种，接收移位脉冲和接收字符的检测脉冲。

- 串行口模式 1 接收数据时的移位脉冲，由定时器 1 的溢出信号和波特率倍增位 SMOD 来共同决定，即由定时器 1 的溢出信号经过 16 或 32 分频得到。
- 接收字符的检测脉冲，其频率是接收移位脉冲的 16 倍。在接收一位数据的时候，有 16 个检测脉冲，以其中的第 7、8、9 这三个脉冲作为真正的接收信号的采样脉冲。对三次采样结果采取三中取二的原则来确定所检测到的值。采样这种机制是为了抑制干扰，由于采样的信号总是在接收位的中间位置，这样便可以避免信号两端的边沿失真，也可以防止由于收发时钟频率不完全一致而带来的错误接收。

在串行口的工作模式 1 中，数据的接收流程如下：

(1) 首先置串行接口控制寄存器 SCON 的 REN=1，启动串行口串行数据接收，RXD 引脚便进行串行口的采样。

(2) 在数据传递的时候 RXD 引脚的状态为 1，当检测到从 1 到 0 的跳变的时候，确认数据起始位 0。

(3) 开始接收一帧的串行数据，在接收移位脉冲的控制下，将收到的数据一位一位地送入移位寄存器，直到 9 位数据完全接收完毕，其中最后一位为停止位。

(4) 当 RI=0，并且接收到的停止位为 1，或者 SM2=0 的时候，8 位数据送入接收缓冲器 SBUF 中，停止位送入 RB8 中，同时置 RI=1；否则，8 位数据不装入 SBUF，放弃当前接收到的数据。

(5) 当数据送入接收缓冲器之后，便可以执行读 SBUF 指令来读取数据，示例如下：

```
MOV    A, SBUF
```

(6) 软件中清零标志位 RI，以便于接收下一次串行数据。

如果采用汇编语言，按照上面的步骤，则其程序示例如下：

```
ORG      0000H  
AJMP MAIN  
  
MAIN:    MOV      SCON, #50H           ;初始化串行口模式 1，允许接收  
         MOV      TMOD, #20H          ;初始化 T1 为定时功能，模式 2  
         MOV      PCON, #80H          ;设置 SMOD=1  
         MOV      TL1, #0F4H          ;波特率 4800bit/s，初值  
         MOV      TH1, #0F4H  
         CLR      ES  
         SETB     TR1  
         JNB      RI, $               ;查询 RI，若 RI=0，表示未接收完毕，等待  
         MOV      A, SBUF  
         CLR      RI  
HERE:    JMP      HERE  
END
```

在该程序中，首先初始化串口为模式 1，允许接收，接着设置波特率，打开定时器。程序中使用 JNB 指令查询 RI，当 RI=1 时，表示串行数据接收完毕，随后将数据读入累加器 A，并清零 RI，以便于接收下一次串行数据。如果采用 C51 语言进行程序设计，按照上面的步骤，则其程序示例如下：

```
#include <reg51.h>           //头文件
```

```
void main(void)                                //主函数
{
    int ch;
    SCON=0x50;                                  //初始化串行口模式 1，允许接收
    TMOD=0x20;                                  //初始化 T1 为定时功能，模式 2
    PCON=0x80;                                  //设置 SMOD=1
    TL1=0xF4;                                   //波特率 4800bit/s，初值
    TH1=0xF4;
    ES=0;                                        //禁止串行中断
    TR1=1;                                       //启动定时器

    while (RI)                                  //等待 RI=1
    {
        ch=SBUF;                                //读取串行数据
        RI=0;                                   //RI 清零
    }
}
```

在该程序中，主函数首先初始化串口为模式 1，允许接收，接着设置波特率，打开定时器。程序中使用 while 循环来查询 RI，当 RI=1 时，表示串行数据接收完毕，随后将数据读入变量 ch，并清零 RI，以便于接收下一次串行数据。

19.5 串行口的工作模式 2

串行口的工作模式模式 2 为固定波特率的串行异步通信方式，可以设置控制寄存器 SCON 的 SM0=1 和 SM1=0 来实现。其波特率不由定时器来设置，只可选 f_{osc}/32 或 f_{osc}/64 两种。在模式 2 中，数据帧的格式，如图 19-13 所示。一帧数据由 11 位构成，按照顺序分别为起始位 1 位、8 位串行数据（低位在前）、可编程位 1 位、停止位 1 位。

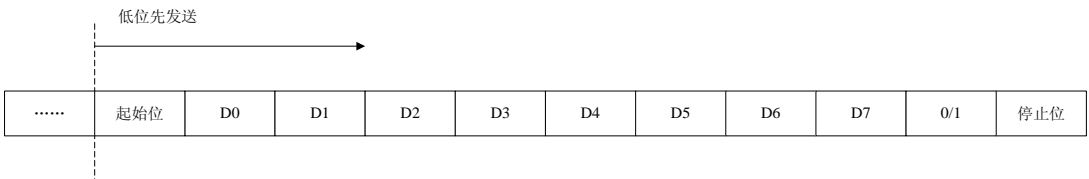


图 19-13 模式 2 的帧格式

19.5.1 模式 2 的发送

在串行口的工作模式 2 中，TXD 引脚为数据发送端。模式 2 的发送共有 9 位有效的数据，在启动发送之前，需要将发送的第 9 位，即可编程位的数值送入寄存器 SCON 中的 TB8 位。这个编程标志位可以由用户自己定义，硬件不做任何规定。例如，用户可以将这一位定义为奇偶校验位或地址/数据标志位。

在串行口的工作模式 2 中，数据的发送流程如下：

- (1) 首先，置串行接口控制寄存器 SCON 的 TI=0，启动串行口发送，并装入 TB8 的值；
- (2) 执行写发送缓冲器 SBUF 指令，示例如下：
MOV SBUF,A
- (3) 硬件自动发送起始位，起始位为逻辑低电平；
- (4) 发送 8 位数据，低位首先发送，高位最后发送；
- (5) 发送第 9 位数据，即 TB8 中的数值；

(6) 硬件自动发送停止位，停止位为逻辑高电平，同时置 TI=1，发送完毕；

(7) 软件清零 TI，以便于下一次串行数据发送。

下面举例讲解串行口模式 2 在程序设计中的应用。如果采用汇编语言，按照上面的步骤，则其程序示例如下：

```

                ORG      0000H
                AJMP     MAIN
MAIN:           MOV      SCON, #80H                ;设置串行口为模式 2
                MOV      PCON, #80H                ;波特率倍增位 SMOD=1
                MOV      A, #46H                    ;读取数据
                MOV      C, P
                MOV      TB8, C                      ;奇偶位送 TB8
                MOV      SBUF, A                     ;发送
DELAY:          JBC      TI, $                       ;判断是否发完
                CLR      TI                          ;清零 TI
HERE:           JMP      HERE
                END
```

在该程序中，采用 TB8 作为奇偶校验位。主程序首先初始化串口，接着将奇偶位送入 TB8，然后将数据发送到 SBUF。当发送完毕的时候，清零 TI，以便于下一次串行数据发送。

如果采用 C51 语言进行程序设计，按照上面的步骤，则其程序示例如下：

```
#include <reg51.h>                                //头文件

void main(void)                                    //主函数
{
    SCON=0x80;                                     //初始化串行口模式 2
    PCON=0x80;                                     //设置 SMOD=1
    ES=0;                                           //禁止串行中断
    TB8=ParityCheck(0x46);                          //奇偶校验位
    SBUF=0x46;
    while(TI)                                       //等待 TI=1
    {
        TI=0;                                       //TI 清零
    }
}
```

在该程序中，采用 TB8 作为奇偶校验位。主程序首先初始化串口，接着判断数据的奇偶位并送入 TB8，然后将数据发送到 SBUF。程序中使用 while 循环查询 TI，当发送完毕的时候，清零 TI，以便于下一次串行数据发送。

19.5.2 模式 2 的接收

在串行口的工作模式 2 中，RXD 引脚为数据接收端。模式 2 的串行数据接收过程和模式 1 基本一致，只不过模式 1 的第 9 位为停止位，而这里则是发送的可编程位。数据接收的过程如下：

(1) 置串行接口控制寄存器 SCON 的 REN=1，启动串行口串行数据接收，引脚 RXD 便进行串行口的采样；

(2) 在数据传递的时候 RXD 引脚的状态为 1，当检测到从 1 到 0 的跳变的时候，确认数据起始位 0；

(3) 开始接收一帧的串行数据，在接收移位脉冲的控制下，将收到的数据一位一位地送入移位寄存器，直至 9 位数据完全接收完毕，其中最后一位为发送的 TB8；

(4) 当 RI=0，且 SM2=0 或接收到的第 9 位数据为 1 时，8 位数据送入接收缓冲器 SBUF 中，第 9 位数据送入 RB8 中，同时置 RI=1；否则，8 位数据不装入 SBUF，放弃当前接收到的

51 单片机开发与应用技术详解

数据；

(5) 当数据送入接收缓冲器之后，便可以执行读 SBUF 指令来读取数据，示例如下：

```
MOV     A, SBUF
```

接收数据真正有效的条件有两个，第一个条件是 RI=0，表示接收缓冲器已空，即 CPU 已把 SBUF 中上次收到的数据读走，可以进行再次写入；第二个条件是 SM=0 或收到的第 9 位数据为 1，根据 SM2 的状态和接收到的第 9 位数据状态来决定接收数据是否有效。

(6) 软件清零 RI，以便于接收下一次串行数据。

在单机通信中，第 9 位一般作为奇偶校验位，应令 SM2=0，以保证可靠的接收；在多机通信时，第 9 位数据一般作为地址/数据标志位，应令 SM2=1，则当第 9 位为 1 的时候，接收的信息为地址帧，串行口将接收该组信息。

下面以第 9 位为奇偶校验位来讲解，如果采用汇编语言进行程序设计，程序示例如下：

```
ORG      0000H
AJMP     MAIN

MAIN:    MOV     SCON, #90H           ;设置串行口为模式 2
         MOV     PCON, #80H          ;波特率倍增位 SMOD=1
         JBC     RI, $                ;判断是否发完
         MOV     A, SBUF
         CLR     RI
         MOV     C, P
         JNC     FUN1                 ;偶校验转向 FUN1
         JNB     RB8, ERR              ;奇校验时 RB8 为 0 则转向 ERR
         SJMP    FUN2
FUN1:    JB      RB8, ERR              ;偶校验时 RB8 为 1 转向 ERR 错误处理
FUN2:    MOV     @R0, A                ;奇偶校验正确的时候存入数据
         INC     R0
ERR:     .....                        ;错误处理部分
HERE:    JMP     HERE
END
```

在本程序中，首先初始化串口，然后接收串行数据，接收完毕后清零 RI，以便于接收下一次串行数据。程序中根据奇偶位是否一致来做相应的处理。

如果采用 C51 语言进行程序设计，按照上面的步骤，则其程序示例如下：

```
#include <reg51.h> //头文件

void main(void) //主函数
{
    int ch;
    SCON=0x90; //初始化串行口模式 2
    PCON=0x80; //设置 SMOD=1
    ES=0; //禁止串行中断
    while(RI) //等待 RI=1
    {
        ch=SBUF;
        RI=0; //RI 清零
    }
    if(RB8== ParityCheck(ch)) //判断奇偶校验位
    {
    }
    else //错误处理
    {
    }
}
```

}

在本程序中，首先初始化串口，然后接收串行数据并保存至变量 **ch**，接收完毕后清零 **RI**，以便于接收下一次串行数据。程序中使用 **if** 判断语句判断奇偶位是否一致，并做相应的处理。

19.6 串行口的工作模式 3

串行口的工作模式 3 为 11 位异步发送接收方式，可以设置控制寄存器 **SCON** 的 **SM0=1** 和 **SM1=1** 来实现。模式 3 和模式 2 的工作方式是一样的，不同的是，模式 3 的波特率由定时器 1 的溢出率和波特率倍增位 **SMOD** 决定，而模式 2 仅有两个固定的波特率可选。这里不再详细介绍，仅给出上面例子按照串口模式 3 工作的程序。

19.6.1 模式 3 的发送

在串行口的工作模式 3 中，**TXD** 引脚为数据发送端。如果采用汇编语言，则数据发送的程序示例如下：

```
ORG      0000H
AJMP     MAIN
MAIN:    MOV     SCON, #0C0H      ;设置串行口为模式 3
        MOV     PCON, #80H      ;波特率倍增位 SMOD=1
        MOV     TL1, #0F4H      ;波特率 4800bit/s，初值
        MOV     TH1, #0F4H
        SETB    TR1             ;启动定时器
MOV      A, #46H               ;读取数据
        MOV     C, P
        MOV     TB8, C          ;奇偶位送 TB8
        MOV     SBUF, A         ;发送
DELAY:   JBC     TI, $           ;判断是否发完
        CLR     TI             ;清零 TI
HERE:    JMP     HERE
END
```

在该程序中，采用 **TB8** 作为奇偶校验位。主程序首先初始化串口，并设置波特率。接着将奇偶位送入 **TB8**，然后将数据发送到 **SBUF**。当发送完毕的时候，清零 **TI**，以便于下一次串行数据发送。

如果采用 **C51** 语言进行程序设计，则其程序示例如下：

```
#include <reg51.h>           //头文件

void main(void)              //主函数
{
    SCON=0xC0;                //初始化串行口模式 2
    PCON=0x80;                //设置 SMOD=1
    TL1=0xF4;                 //波特率 4800bit/s，初值
    TH1=0xF4;
    ES=0;                     //禁止串行中断
    TR1=1;                    //启动定时器
    TB8=ParityCheck(0x46);    //奇偶校验位
    SBUF=0x46;
    while(TI)                 //等待 TI=1
    {
        TI=0;                 //TI 清零
    }
```

```
}
}
```

在该程序中，采用 **TB8** 作为奇偶校验位。主程序首先初始化串口，并设置波特率，启动定时器。接着判断数据的奇偶位并送入 **TB8**，然后将数据发送到 **SBUF**。程序中使用 **while** 循环查询 **TI**，当发送完毕的时候，清零 **TI**，以便于下一次串行数据发送。

19.6.2 模式 3 的接收

在串行口的工作模式 3 中，**RXD** 引脚为数据接收端。如果采用汇编语言进行程序设计，则数据接收的程序示例如下：

```

        ORG      0000H
        AJMP     MAIN
MAIN:    MOV      SCON,#0D0H          ;设置串行口为模式 2
        MOV      PCON,#80H          ;波特率倍增位 SMOD=1
        MOV      TL1,#0F4H          ;波特率 4800bit/s , 初值
        MOV      TH1,#0F4H
        SETB     TR1                ;启动定时器
        JBC      RI,$                ;判断是否发完
        MOV      A,SBUF
        CLR      RI
        MOV      C,P
        JNC      FUN1               ;偶校验转向 FUN1
        JNB      RB8,ERR             ;奇校验时 RB8 为 0 则转向 ERR
        SJMP     FUN2
FUN1:    JB       RB8,ERR             ;偶校验时 RB8 为 1 转向 ERR 错误处理
FUN2:    MOV      @R0,A              ;奇偶校验正确的时候存入数据
        INC      R0
ERR:     .....                      ;错误处理部分
HERE:    JMP      HERE
        END
```

在本程序中，首先初始化串口允许接收，并设置波特率，启动定时器。然后接收串行数据，接收完毕后清零 **RI**，以便于接收下一次串行数据。程序中根据奇偶位是否一致来做相应的处理。如果采用 **C51** 语言进行程序设计，则其程序示例如下：

```
#include <reg51.h>                //头文件

void main(void)                    //主函数
{
    int ch;
    SCON=0xD0;                     //初始化串行口模式 2
    PCON=0x80;                     //设置 SMOD=1
    TL1=0xF4;                      //波特率 4800bit/s , 初值
    TH1=0xF4;
    ES=0;                          //禁止串行中断
    TR1=1;                         //启动定时器
    while(RI)                      //等待 RI=1
    {
        ch=SBUF;
        RI=0;                     //RI 清零
    }
    if(RB8== ParityCheck(ch))      //判断奇偶校验位
    {
```

```
}
else                                     //错误处理
{
}
}
```

在本程序中，首先初始化串口允许接收，并设置波特率，启动定时器。然后接收串行数据并保存至变量 `ch`，接收完毕后清零 `RI`，以便于接收下一次串行数据。程序中使用 `if` 判断语句判断奇偶位是否一致，并做相应的处理。

19.7 双机通信程序设计

双机通信是利用单片机的串行接口，实现单片机与另一个单片机的点对点的异步串行通信。单片机之间的串行通信比较简单，由于接口电平完全一致，因此，只要将两个单片机的发送和接收引脚交叉相接即可，如图 19-14 所示。

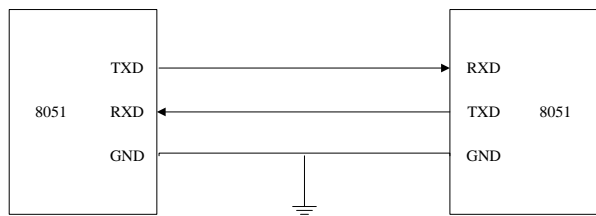


图 19-14 单片机之间的双机通信

下面分别采用查询方式和中断方式来进行单片机之间的双机通信。

19.7.1 查询方式

假设发送方 A 需要把片内 RAM 中 50H~6FH 单元中的数据，通过串行接口发送给接收方 B。接收方 B 将接收到的这 32 个字节数据后，存入片外 1000H~101FH。

发送方和接收方均采用 8051 单片机，外接 $f_{osc}=6\text{MHz}$ 的晶振，使用串行口工作方式 2，波特率规定为 187.5kbit/s，需要使用波特率倍增位，即置 `SMOD=1`。

发送方定义 `TB8` 作为奇偶校验位。接收方进行奇偶校验位 `RB8` 的判断，如果出错，则置 `F0` 标志位为 1；如果正确，则置 `F0` 标志位为 0，然后返回。发送方 A 的程序示例如下：

```
START:  MOV     SCON,#80H           ;设置串行口为模式 2
        MOV     PCON,#80H          ;波特率倍增位 SMOD=1
        MOV     R7,20H             ;设置数据块长度
        MOV     R1,#50H            ;设置数据块指针
SED:    MOV     A,@R1              ;读取数据
        MOV     C,P
        MOV     TB8,C              ;奇偶位送 TB8
        MOV     SBUF,A             ;发送
DELAY:  JBC     TI,CONT             ;判断是否发完
        AJMP    DELAY
CONT:   INC     R1                  ;R1 加 1
        DJNZ    R7,SED             ;循环发送，至全部发送完毕
        RET                        ;返回
```

该程序中，首先初始化串口为模式 2，通过设置波特率倍增位 `SMOD=1` 来指定波特率。接着设置发送数据块的长度及首地址。在发送数据前，将数据的奇偶位送入 `TB8`，然后循环发送直至数据全部发送完毕。接收方 B 的程序示例如下：

```
START:  MOV      SCON, #80H           ;设置串行口为模式 2
        MOV      PCON, #80H         ;置 SMOD=1
        MOV      R6, #20H           ;设置接收数据块长度
        MOV      DPTR, #1000H       ;设置接收数据块指针
        SETB     REN                ;允许接收
DELAY:   JBC      RI, READ            ;判断一帧是否接收完
        AJMP     DELAY
READ:    MOV      A, SBUF             ;读数据
        JNB      PSW.0, PZ
        JNB      RB8, ERR
        SJMP     RIGHT
PZ:      JB       RB8, ERR            ;出错
RIGHT:   MOVX     @DPTR, A            ;正确，存放数据
        INC      DPTR               ;修改指针
        DJNZ     R6, DELAY           ;判断数据块是否接收完毕
        CLR      PSW.5              ;接收正确，且接收完，置 F0=0
        RET
ERR:     SETB     PSW.5              ;如果出错，置 F0=1
        RET
```

该程序中，首先初始化串口为模式 2，通过设置波特率倍增位 SMOD=1 来指定波特率。接着设置接收数据块的长度及首地址。在接收数据后，判断奇偶位，如果出错则置 F0 标志位为 1。

19.7.2 中断方式

采用中断方式接收串行数据，可以提高 CPU 的工作效率。

假设发送方 A 将外部数据存储器 1000H~10FFH 单元中的数据，通过串行接口发送给接收方 B。接收方 B 接收 A 发送的数据，并写入以 2000H 为首地址的外部数据存储器中。

发送方 A 在发送数据之前，首先将数据块的长度发送过去，当发送完 256 个字节后，再发送一个累加校验和。接收方 B 首先接收数据长度，然后接收 256 个字节的数据，再接收校验码，进行累加和校验，数据发送结束后，向发送方 A 发送一个状态字，表示正确或出错，出错则要求重发。接收方采用中断的方式，设置两个标志位来判断接收到的信息是数据块长度、数据、还是校验和。

发送方和接收方均采用 8051 单片机，外接 f_{osc}=6MHz 的晶振，串行口工作于模式 1，定时器 T1 按方式 1 工作，波特率规定为 2400bit/s，需要使用波特率倍增位，即置 SMOD=1。发送方 A 的程序示例如下：

```
START:  MOV      TMOD, #20H          ;定时器设置
        MOV      TH1, #0F3H         ;赋初值
        MOV      TL1, #0F3H
        SETB     TR1                ;启动 T1
        MOV      SCON, #50H         ;串口初始化
        MOV      PCON, #80H         ;波特率倍增位 SMOD=1
REP:     MOV      DPTR, 1000H        ;数据指针首地址
        MOV      R4, #00H           ;校验和寄存器初始化
        MOV      R5, #00H           ;长度寄存器初始化
        MOV      SBUF, R5           ;发送长度
FUN1:    JBC      TI, FUN2            ;等待发送
        AJMP     FUN1
```

```
FUN2:  MOVX    A,@DPTR      ;读数据
      MOV     SBUF,A      ;发送数据
      ADD     R4,A        ;构造校验和
      INC     DPTR
FUN4:  JBC     TI,FUN3
      AJMP    FUN4
      DJNZ    R5,FUN2     ;判断是否发送完
      MOV     SBUF,R4     ;发送校验码
FUN6:  JBC     TI,FUN5
      AJMP    FUN6
FUN5:  JBC     RI,FUN7     ;等待 B 机回答
      AJMP    FUN5
FUN7:  MOV     A,SBUF
      JZ      FUN8        ;发送正确返回
      AJMP    REP        ;发送出错，重发
FUN8:  RET
```

该程序中，发送方仍采用查询方式来发送数据。主程序首先初始化定时器 T1，并启动定时器，然后初始化串口。在发送数据时首先发送数据块的长度，然后循环发送数据块中的每一位数据及校验码。最后等待接收方 B 回答，如果发送出错则重新发送。接收方 B 的程序示例如下：

```
      ORG     0000H
      LJMP    CHU          ;转向初始化程序
      ORG     0023H        ;串行口中断入口地址
      LJMP    INTC        ;转向中断服务程序
      ORG     1000H
CHU:   MOV     TMOD,#20H    ;定时器设置
      MOV     TH1,#0F3H
      MOV     TL1,#0F3H
      SETB    TR1          ;启动 T1
      MOV     SCON,#50H    ;串口初始化
      MOV     PCON,#80H    ;波特率倍增位 SMOD=1
      SETB    7FH
      SETB    7EH        ;标志位初始化置 1
      MOV     41H,#20H     ;接收数据在外部 RAM 中的地址
      MOV     40H,#00H
      MOV     44H,#00H
      SETB    EA          ;开中断
      SETB    ES          ;允许串行口中断
      LJMP    MAIN
      .....
INTC:  CLR     EA          ;中断服务程序，关中断
      CLR     RI          ;清除中断标志
      PUSH    DPH         ;保护现场
      PUSH    DPL
      PUSH    A
      JB      7FH,LEN      ;判断
      JB      7EH,DATA     ;判断
SUM:   MOV     A,SBUF      ;接收校验和
      CJNE    A,44H,ERR    ;判断发送正确与否
RIGHT: MOV     A,#00H
```

51 单片机开发与应用技术详解

```

                                MOV     SBUF,A                ;正确，发状态 00H
DELAY1:  JNB     TI,DELAY1
                                CLR     TI
                                SJMP    RETURN

ERR:      MOV     A,#0FFH                ;出错，发状态 FFH
                                MOV     SBUF,A
DELAY2:  JNB     TI,DELAY2
                                CLR     TI
                                SJMP    AGAIN

LEN:      MOV     A,SBUF
                                MOV     42H,A

                                CLR     7FH                ;清长度标志
                                SJMP    RETURN

DATA:    MOV     DPH,41H
                                MOV     DPL,40H

                                MOV     A,SBUF                ;接收数据
                                MOVX    @DPTR,A
                                INC     DPTR
                                MOV     41H,DPH
                                MOV     40H,DPL

                                ADD     A,44H                ;构造累加和
                                MOV     44H,A
                                DJNZ    42H,RETURN

                                CLR     7EH                ;数据标志位清零
                                SJMP    RETURN

AGAIN:   SETB    7FH
                                SETB    7EH

                                MOV     44H,#00H            ;清零累加和寄存器
                                MOV     41H,#20H
                                MOV     40H,#00H

RETURN:   POP     A                ;恢复现场
                                POP     DPL
                                POP     DPH

                                SETB    EA                ;开中断
                                RETI
                                END
```

该程序中，接收方采用中断方式来接收数据。主程序首先初始化定时器 T1，并启动定时器，然后初始化串口。首先接收发送方的数据块长度，然后循环接收数据块中的每一个数据以及校验码。最后根据校验数据向发送方做出应答，如果出错则要求发送方重新发送。

19.8 多机通信程序设计

多机通信是利用单片机的串行接口，实现单片机与另外多个单片机的异步串行通信。一般采用主从式多机通信方式，如图 19-15 所示。在这种方式中，有一台主机和多台从机。主机可以向各个从机或指定的从机发送信息，各个从机发送的信息只能被主机接收，从机之间不能进行通信。

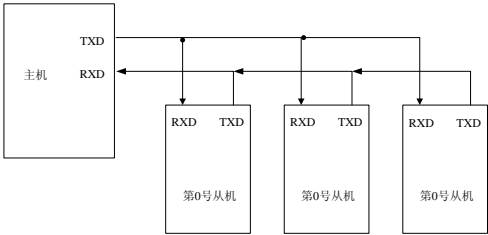


图 19-15 单片机之间的多机通信

19.8.1 多机通信原理

单片机串行多机通信必须使用串行口的模式 2 和模式 3。在多机通信中，为了保证主机能够正确识别所选择的从机并进行通信，主、从机需要正确地设置和判断多机通信控制位 SM2 和发送接收的第 9 位数据，即 TB8 或 RB8。

主机在发送信息时，依靠 TB8 标志位来区分发送的信息为地址信息还是数据命令信息。当设置 TB8=1 时，发送的是地址信号；当设置 TB8=0 时，发送的是数据或命令。

从机主要依靠 SM2 标志位的设置实现对主机的响应。当从机的 SM2=1 时，该从机只接收地址帧，RB8=1，对数据帧 RB=0 不进行处理；当从机的 SM2=0 时，该从机接收所有主机发送的信息。

单片机多机通信的过程如下：

- (1) 所有的从机复位，置 SM2=1，使其处于准备接收一帧地址数据的状态；
- (2) 主机设置第 9 位 TB8=1，发送一帧地址信息，与所需的从机进行联系；
- (3) 各个从机接收到地址信息后，分别与自己的地址进行比较。对于地址相符的从机，置 SM2=0，以便于接收主机随后发出的所有信息；对于地址不符的从机，仍保持 SM2=1，对主机随后发送的数据不进行处理。
- (4) 主机发送控制指令和数据给被寻址的从机，此时一帧数据的第 9 位 TB8=0。
- (5) 当从机接收数据结束的时候，置 SM2=1，返回接收地址帧状态。
- (6) 主机继续发送其他地址帧，与其他从机进行通信。

19.8.2 多机通信协议约定

主机为了能够正确地和指定的从机正确通信，双方需要对各种地址、数据、指令和状态等进行明确的约定，即通信的主从机之间需要具有明确的通信协议。下面给出一种常用的协议约定。

- 主机的控制命令：00H 表示主机要求从机接收数据块；01H 表示主机要求从机发送数据块。其他的保留，可自定义用途。
- 从机的地址定义在 00H~FEH 之间，即系统中最多允许连接 255 个从机。对于地址 FFH，定义为对所有从机都有效的控制命令，用于将各从机恢复到复位状态，即 SM=1。
- 从机的状态字格式，如图 19-16 所示。其中，ERR=1，表示从机接收到非法的命令；如果 TRDY=1，表示从机发送准备完毕；如果 RRDY=1，表示从机接收准备完毕。

D7	D6	D5	D4	D3	D2	D1	D0
ERR	0	0	0	0	0	TRDY	RRDY

图 19-16 从机的状态字

按照上面的协议约定，主机首先发送地址帧，被寻址的从机向主机回送本地地址。主机判断地址是否相符，然后给被寻址的从机发送控制命令，被寻址的从机根据其命令向主机发送本地机的状态。若主机判断状态正常，即开始发送或接收数据，发送或接收的第一个字节为数据块长度，后面紧跟的便是数据。若主机判断从机的状态不正常，需要重新联系。

19.8.3 多机通信程序设计

下面举例讲解多机通信的程序设计。假设主机和从机均采用串行接口的工作模式 2，外接 11.0592MHz 的晶振，波特率为 2400bit/s。

主机中使用的 5 个寄存器，其中，R0 为主机接收数据块首地址；R1 为主机发送数据块首地址；R2 为被寻址的从机地址；R3 为主机发出的控制指令；R4 为数据块长度。在执行通信前，应该首先在 R0~R4 中预置好参数。

这里将主机程序封装成子程序的形式，需要进行通信的时候，直接调用即可。主机部分的子程序流程图，如图 19-17 所示。程序代码示例如下：

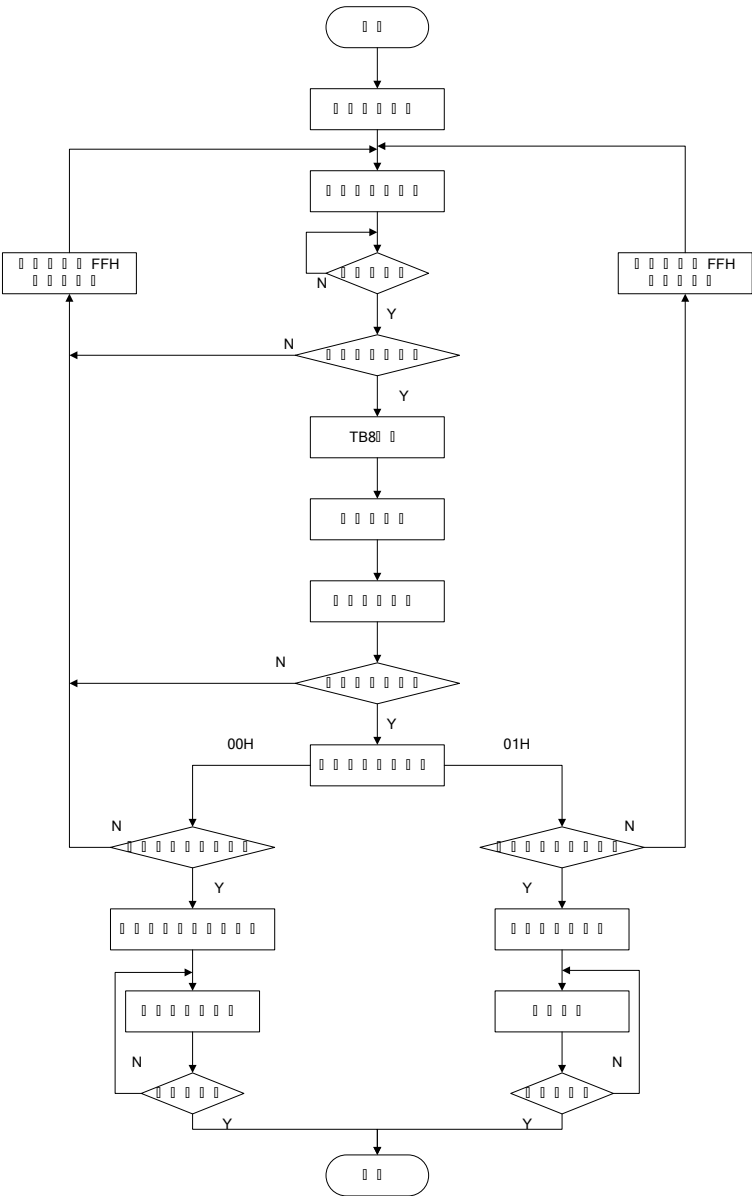


图 19-17 主机程序流程图

NPC:

MOV

TMOD, #20H

;定时器计数器 T1 初始化为定时功能，模式 2

MOV

TL1, #0F3H

;T1 赋初值

MOV

TH1, #0F3H

SETB

TR1

;启动定 T1

MOV

PCON, #80H

;设置波特率倍增位 SMOD=1

MOV

SCON, #0D8H

;设置串行口模式 3

NPC1:

MOV

SBUF, R2

;寻址从机

JNB

TI, \$

;等待

CLR

TI

;发送完，清零 TI

DELAY1:

JBC

RI, NPC2

;等待应答

	SJMP	DELAY1	
NPC2:	MOV	A, SBUF	; 读取从机的应答地址
	XRL	A, R2	; 进行地址核对
	JZ	NPC4	; 如果地址正确, 则转 NPC4 继续执行
NPC3:	SETB	TB8	; 如果地址不符, 重新联系
	MOV	SBUF, #0FFH	; 给从机发送复位命令, 即 TB 置 1
	JNB	TI, \$; 等待
	CLR	TI	; 清零 TI
	SJMP	NPC1	; 转重发地址
NPC4:	CLR	TB8	; 地址符合, TB8 清零, 准备发送数据/命令
	MOV	SBUF, R3	; 给从机发送命令
	JNB	TI, \$	
	CLR	TI	
DELAY2:	JBC	RI, NPC5	; 等待从机应答
	SJMP	DELAY2	
NPC5:	MOV	A, SBUF	; 读取从机的应答信息
	JNB	ACC.7, NPC6	; 核对命令, 正确则转
	SJMP	NPC3	; 如果出错, 转重新联系
NPC6:	CJNE	R3, #00, NPC7	; 若要求从机发送, 则转 NPC7
	JNB	ACC.0, NPC3	; 要求从机接收, 从机未准备好, 重新联系
STX:	MOV	SBUF, R4	; 从机准备好, 向从机发送数据块长度
DELAY3:	JBC	TI, STX1	; 发送结束, 则转
	SJMP	DELAY3	; 未发送完, 等待
STX1:	MOV	SBUF, @R0	; 向从机发送数据
	JNB	TI, \$	
	CLR	TI	
	INC	R0	; 修改地址, 指向下一个地址单元
	DJNZ	R4, STX1	; 数据未发送完, 继续发送
	RET		; 数据发送完毕, 返回主程序
NPC7:	JNB	ACC.1, NPC3	; 若从机发送未准备好, 转重新联系
SRT:	JNB	RI, \$; 等待接收从机发来的数据块长度
	CLR	RI	; 清零 TI 位
	MOV	A, SBUF	; 取出收到的数据
	MOV	R4, A	; 数据块长度送计数器 R4
	MOV	@R1, A	; 数据块长度存入数据存储区
	INC	R1	; 修改地址
SRX1:	JNB	RI, \$; 等待接收从机数据
	CLR	RI	
	MOV	@R1, SBUF	; 接收的数据存入数据存储区
	INC	R1	; 修改地址, 指向下一个地址单元
	DJNZ	R4, SRX1	; 数据未接收完, 继续接收
	RET		; 返回主程序

该程序中, 首先初始化串口为模式 3, 并设置波特率和启动定时器。接着主机采用查询的方式进行通信, 首先发送地址帧, 来寻址从机。当从机的应答地址相符后, 便发送命令帧, 根据从机的返回状态来决定是否发送或接收数据。该程序在通信过程中充分考虑的错误可能性, 当出错时将返回到初始状态进行重新联系。

第 19 章 51 系列单片机的串行接口

```

MOV     SCON, #0F0H      ; 串行口为模式 3, 允许接收, SM2=1
MOV     PCON, #80H       ; 波特率倍增位 SMOD=1
MOV     08H, #40H        ; 发送数据的首地址送入 R0
MOV     09H, #60H        ; 接收数据的首地址送入 R1
SETB    EA               ; 开启所有中断
SETB    ES               ; 允许串行口中断
LJMP    ZHU              ; 转主程序, 这里暂未给出
.....
MORB:   CLR     RI        ; 清零中断申请 RI
        PUSH    ACC       ; 保护现场
        PUSH    PSW
        CLR     RS1
        SETB    RS0
        MOV     A, SBUF    ; 读取主机发来的地址
        XRL     A, #12H    ; 核对本机地址
        JZ      MORB1     ; 地址符合, 则转向 MORB1
RETURN:  SETB    SM2       ; SM2=1
        POP     PSW       ; 恢复现场
        POP     ACC
        RETI
MORB1:  CLR     SM2       ; 置 SM2=0, 准备接收数据/命令
        CLR     TI        ; 清零 TI
        MOV     SBUF, #12H ; 向主机发回本机地址供核对
        JNB     TI, $      ; 等待发送结束
        CLR     TI        ; 清零 TI
        JNB     RI, $      ; 等待主机发送数据/命令
        CLR     RI        ; 清零 RI
        JNB     RB8, MORB2 ; 是数据/命令, 则跳转
        SJMP    RETURN    ; RB8=1 是复位信号, 转返回
MORB2:  MOV     A, SBUF    ; 取出命令
        CJNE    A, #02H, NEXT ; 检查命令是否合法
NEXT:   JC      MORB3      ; (A) <= 01H, 是合法命令, 跳转
        MOV     SBUF, #80H ; 是非法命令, 向主机发出状态字 ERR=1
        JNB     TI, $      ; 等待
        CLR     TI        ; 清零 TI
        SJMP    RETURN    ; 转返回
MORB3:  JZ      CMOD       ; 转接收
CMD1:   JB      PSW.1, MORB4 ; 转发送
        MOV     SBUF, #00H ; 未准备好, 向主机发出状态字 TRDY=0
        JNB     TI, $      ; 等待
        CLR     TI        ; 清零 TI
        SJMP    RETURN    ; 转返回
MORB4:  SBUF    #02H       ; 向主机发出发送准备就绪信号
        JNB     TI, $      ; 等待
        CLR     TI        ; 清零 TI
        CLR     PSW.1

```

51 单片机开发与应用技术详解

```

MOV      R4,@R0      ;数据块长度送入 R4
INC      R4           ;R4 加 1
LOOP1:   MOV      SBUF,@R0      ;发送数据 ( 第一个字节是数据块长度 )
JNB      TI,$
CLR      TI           ;清零 TI
INC      R0           ;修改地址, 指向下一个地址单元
MOV      R4,LOOP1     ;数据未发送完, 继续
LJMP     RETURN       ;数据发送完, 则转
CMOD:    JB      PSW.5,MORB5    ;若 PSW.5=1, 转接收
MOV      SBUF,#00H    ;未准备好, 向主机发出 RRDY=0 的状态字
JNB      TI,$
CLR      TI           ;清零 TI
SJMP     RETURN       ;转返回
MORB5:   MOV      SBUF,#01H    ;向主机发出接收准备就绪信号
JNB      TI,$
CLR      TI           ;清零 TI
CLR      PSW.5
JNB      RI,$         ;等待接收数据块长度
CLR      RI           ;清零 RI
MOV      A,SBUF       ;读取数据块长度
MOV      @R1,A        ;数据块长度送内存
INC      R1           ;地址指针加 1, 指向下一个单元
MOV      R4,A         ;数据块长度送 R4
LOOP2:   JNB      RI,$         ;等待接收数据
CLR      RI           ;清零 RI
MOV      @R1,SBUF     ;读取接收的数据送内存
INC      R1           ;指向下一个地址单元
MOV      R4,LOOP2
LJMP     RETURN       ;转返回
END
```

该程序中，首先初始化串口为模式 2，并设置波特率和启动定时器。接着判断接收到的地址是否和本机地址相符，如果相符则向主机返回本机地址并等待接收下一帧信息。如果主机发出控制命令，从机判断该命令并开始发送或接收数据。该程序在通信过程中对可能的错误进行了充分处理。

19.9 小结

本章首先详细介绍了串行通信的基本方式，包括异步串行通信和同步串行通信，以及单工制式、半双工制式和全双工制式三种数据传送方式。51 系列单片机集成了全双工的串行接口，本章详细介绍了单片机串行接口的内部结构、程序控制、4 种工作模式及其程序设计等。最后介绍了单片机的串行接口在双机通信和多机通信方面的应用。单片机串行口的应用非常广泛，熟练掌握本章内容很重要。

第 20 章 C51 下的 RTX-51 实时多任务操作系统

RTX-51 是适用于 51 系列单片机的一种微处理器实时多任务操作系统 (RTOS)。在一般性的小型单片机程序中, 采用单一进程并配以中断处理则可以完成大部分的设计需求。但是, 对于一些复杂的设计项目, 需要同时执行多个任务或进程。此时, 传统的设计方式使得程序比较复杂, 而且性能难以满足要求。RTX-51 是一个强大的任务管理工具, 其可以在单个 51 系列 CPU 上管理多个任务或进程。RTX-51 使得复杂的多任务程序设计变得简单。

本章将主要介绍 RTX-51 实时多任务操作系统, 包括 RTX-51 的种类、RTX-51 的任务管理及配置、系统函数等。本章最后将通过一个简单的实例来讲解 RTX-51 实时多任务操作系统的设计。

20.1 RTX-51 实时多任务操作系统简介

在复杂的微处理应用项目中, 需要 CPU 同时执行多个任务或进程。实时多任务操作系统 (RTOS) 可以灵活地为各个任务分配系统资源, 包括 CPU、内存及时间等。在 51 系列单片机中, 可以运行 RTX-51 多任务实时操作系统。

RTX-51 实时多任务操作系统使用标准的 C51 语言来编写程序, 可以运行于所有的 51 系列单片机中。RTX-51 自身提供了灵活的时间分配, 以及任务的响应和切换。

20.1.1 RTX-51 种类

目前, 在 Keil μ Vision3 中支持的 RTX-51 有两个版本, 即 RTX-51 FULL 和 RTX-51 Tiny。下面分别介绍。

- RTX-51 FULL 在运行时, 允许多达 4 个优先权任务的切换和循环, 并能够并行地利用单片机的中断功能。RTX-51 FULL 还支持程序中的信号传递, 以及与系统邮箱 (Mailbox System) 和信号量之间的消息传递。RTX-51 FULL 中的 `os_wait` 函数可以支持中断、超时、中断或任务的信号、中断或任务的消息及信号量的事件等待。同时, RTX-51 FULL 中还可以进行存储器分配及释放。
- RTX-51 Tiny 是 RTX-51 FULL 的一个精简子集, 主要运行在没有外部存储器扩展的 51 单片机系统中。但是, 使用 RTX-51 Tiny 也可以访问外部的存储器。RTX-51 Tiny 同样允许任务的切换、中断功能的并行应用及信号传递。RTX-51 Tiny 的 `os_wait` 函数可以支持超时、时间间隔及来自中断或任务的信号等待事件。在 RTX-51 Tiny 中, 不能进行占先式任务处理、消息处理及存储器的分配和释放。

为了实现 RTX-51 的支持, 需要在程序的头文件列表中加入 RTX51.H 或 RTX51TNY.H。同

51 单片机开发与应用技术详解

时，需要在 Keil μ Vision3 的集成开发环境中指定目标操作系统，操作步骤如下：

(1) 选择“Project”→“Options for Target ‘Target 1’”命令，此时弹出“Options for Target ‘Target 1’”对话框，如图 20-1 所示。

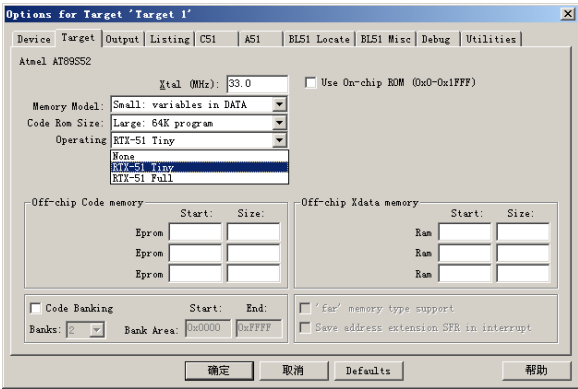


图 20-1 选择目标操作系统

(2) 在“Target”选项卡中，打开“Operating”下拉菜单，从中可以选择 RTX-51 Tiny 或 RTX-51 FULL 操作系统。

(3) 单击“确定”按钮，关闭对话框，完成目标操作系统的指定。

当在 Keil μ Vision3 中选择目标操作系统之后，Keil μ Vision3 内部的链接器将自动为该项目添加合适的 RTX-51 库文件。

20.1.2 RTX-51 与单任务程序的比较

对于不同的设计任务，可以采用不同的方式来完成。前面几章介绍的 C51 语言程序，均包含 main 主函数。程序从 main 函数开始执行，是一个单一的进程，除非有中断打断；而 RTX-51 多任务操作系统的程序，则允许多个进程并行执行。

1. 简单的单任务程序

简单的单任务程序，采用一个 main 主函数开始，程序中使用无限循环保证程序的连续执行。简单的单任务程序示例如下。在该程序的 while 循环中，不断地将 count 加 1，然后通过串口输出。

```
#include <reg52.h>
#include <stdio.h>

int count;

void main() //主函数
{
    while(1) //主循环
    {
        count++; //计数器加 1
        printf("count=%d",count); //输出
    }
}
```

2. 多任务循环程序

多任务循环程序仍然采用 main 函数开始，使用无限循环保证程序的连续执行。为了实现多个任务的操作，则需要在程序中按照一定的顺序切换任务的执行，或者采用中断系统。一个

典型的多任务循环程序示例如下。在程序中，两个计数器循环加 1 并通过串口输出。

```
#include <reg52.h>
#include <stdio.h>

int count0;
int count1;

void main()                                //主函数
{
    while(1)                                //主循环
    {
        count0++;                            //计数器 count0 加 1
        printf("count0=%d",count0);
        count1++;                            //计数器 count1 加 1
        printf("count1=%d",count1);
    }
}
```

3. RTX-51 多任务程序

RTX-51 多任务程序执行多个任务或进程的调度，允许多个循环或任务准并行执行。RTX-51 内核将有效的 CPU 时间划分为时间片，然后将时间片合理地分配给多个任务。程序中每个任务执行预定数量的时间，然后切换到另一个任务的时间片上执行。RTX-51 在任务切换的时候非常短暂，因此可以实现准并行多任务执行。当然 RTX-51 多任务系统执行的速度在很大程度上依赖于单片机的 CPU 频率。对于上面的多任务循环程序，同样可以采用 RTX-51 来实现，示例如下：

```
#include <RTX51TNY.h>
#include <reg52.h>
#include <stdio.h>

int count0;
int count1;

Thread0 () _task_ 0                        //任务 0
{
    os_create_task(1);                      //创建任务 1
    while(1)
    {
        printf("count0=%d\n",count0++);    //打印输出 count0
    }
}

Thread1 () _task_ 1                        //任务 1
{
    while(1)
    {
        printf("count1=%d\n",count1++);    //打印输出 count1
    }
}
```

在该程序中，定义了两个任务，任务 0 的执行函数为 Thread0，任务 1 的执行函数为 Thread1。RTX-51 从任务 0 开始执行程序，并在任务 0 中创建任务 1 为准备执行。当任务 0 的时间片执行完毕后，RTX-51 内核转向任务 1 开始执行另一个时间片。任务 1 的时间片结束后，则重新返回任务 0 执行。此 RTX-51 程序执行时，两个任务无限循环切换，实现准并行执行。

20.2 RTX-51 的任务调度

RTX-51 实时多任务操作系统的程序结构和以前介绍的标准单进程 C51 语言程序有很大的区别。本节将主要介绍 RTX-51 多任务程序的 4 种任务调度方式，分别为循环任务调度、事件任务调度、信号任务调度和抢先任务切换。另外，还将介绍 RTX-51 的一些其他的特性。

20.2.1 RTX-51 循环任务调度

RTX-51 循环任务调度是按照预先划分的时间片来循环轮流执行多个任务的。在 RTX-51 中，使用一个定时子程序，其中断驱动由单片机的硬件定时器来完成。这个定时子程序产生周期性的中断，用来驱动 RTX-51 时钟实现时间片的划分。在 RTX-51 循环任务调度中，程序不要求有 main 主函数。RTX-51 内核自动从任务 0 来开始执行。上一节中的 RTX-51 多任务程序便是采用的这种方式。

对于某些多任务程序，如果包含 main 主函数，则需要使用 os_create_task (RTX-51 Tiny) 和 os_start_system (RTX-51 FULL) 函数来人工启动 RTX-51 实时操作系统。

20.2.2 RTX-51 事件任务调度

RTX-51 事件任务调度是使用事件来实现多任务之间切换的调度方式。RTX-51 的事件可以用来更加灵活地为各个任务分配 CPU 时间。在 RTX-51 系统中，可以使用 os_wait 函数向内核发送事件，暂停当前的任务，从而可以实现在等待指定的事件时可以执行其他的任务。最典型的 RTX-51 事件是时钟信号的超时周期。在 RTX-51 系统中，使用 os_wait 函数发送中断事件的程序示例如下：

```
#include <RTX51TNY.h>
#include <reg52.h>
#include <stdio.h>

int count0;
int count1;

Thread0 () _task_ 0 //任务 0
{
    os_create_task(1); //创建任务 1
    while(1)
    {
        printf("count0=%d\n",count0++); //打印输出 count0
        os_wait(K_TMO,2); //暂停两个信号
    }
}

Thread1 () _task_ 1 //任务 1
{
    while(1)
    {
        printf("count1=%d\n",count1++); //打印输出 count1
        os_wait(K_TMO,3); //暂停三个信号
    }
}
```

在该程序中，定义了两个任务，任务 0 的执行函数为 Thread0，任务 1 的执行函数为 Thread1。RTX-51 从任务 0 开始执行程序，并在任务 0 中创建任务 1 为准备执行。当 count0 计数加 1 并输出后，被 os_wait 函数中断，任务 0 将暂停两个报时信号。此时，RTX-51 内核将切换到任务

1 来执行。当 count1 计数加 1 并输出后，任务 1 同样被 os_wait 函数中断，任务 1 将暂停三个报时信号。此时 RTX-51 内核进入空闲状态，没有任务可以执行。当两个报时信号过去后，任务 0 才可以重新执行。

因此，此程序运行的结果是，每过两个报时信号 count0 计数加 1 并输出，每过三个报时信号 count1 计数加 1 并输出。

20.2.3 RTX-51 信号任务调度

RTX-51 信号任务调度是使用信号来完成多任务之间切换的调度方式。RTX-51 的信号同样用来调度系统中的多任务。在 RTX-51 系统中，可以使用 os_send_signal 函数向另一个任务发送信号，接收信号的任务使用 os_wait 函数等待信号。当任务接收到信号后，便结束等待状态，开始向下执行。如果任务在使用 os_wait 函数等待接收信号之前，信号已经发送过来，那么该任务将立即继续执行，不会再等待。使用 RTX-51 的信号进行任务调度的程序示例如下：

```
#include <RTX51TNY.h>
#include <reg52.h>
#include <stdio.h>

int count0;
int count1;

Thread0 () _task_ 0                                //任务 0
{
    os_create_task(1);                               //创建任务 1
    while(1)
    {
        printf("count0=%d\n",count0++);             //打印输出 count0
        if(count0==10)
        {
            os_send_signal(1);                       //向任务 1 发送信号
            count0=0;
        }
    }
}

Thread1 () _task_ 1                                //任务 1
{
    while(1)
    {
        os_wait(K_SIG,0,0);                         //等待信号
        printf("count1=%d\n",count1++);             //打印输出 count1
    }
}
```

在该程序中，定义了两个任务，任务 0 的执行函数为 Thread0，任务 1 的执行函数为 Thread1。RTX-51 从任务 0 开始执行程序，并在任务 0 中创建任务 1 为准备执行。而任务 1 则在一开始便使用 os_wait 函数等待其他任务的信号；而任务 0 则在 count0 计数达到 10 之后，向任务 1 发送信号，并重新置 count0 为 0。任务 1 接收到信号并到下一个定时信号产生时，便将 count1 计数加 1 并输出。

20.2.4 优先级及抢先任务切换

在 RTX-51 FULL 中提供了优先级设置及抢先的任务切换，而 RTX-51 Tiny 中则不具备这个功能。在默认情况下，RTX-51 所有的任务优先级均为 0，这是系统中最低的优先级。RTX-51

FULL 中优先级可以设置为 0~3。当高优先级的任务生效时，RTX-51 将中断低优先级的任务执行。使用优先级进行抢先任务切换的程序示例如下：


```
#include <RTX51.h>
#include <reg52.h>
#include <stdio.h>

int count0;
int count1;

Thread0 () _task_ 0 //任务 0
{
    os_create_task(1); //创建任务 1
    while(1)
    {
        printf("count0=%d\n",count0++); //打印输出 count0
        if(count0==10)
        {
            os_send_signal(1); //向任务 1 发送信号
            count0=0;
        }
    }
}

Thread1 () _task_ 1 _priority_ 1 //任务 1
{
    while(1)
    {
        os_wait(K_SIG,0,0); //等待信号
        printf("count1=%d\n",count1++); //打印输出 count1
    }
}
```

在该程序中，定义了两个任务。任务 0 的执行函数为 Thread0，优先级默认为 0。任务 1 的执行函数为 Thread1，优先级设定为 1。RTX-51 从任务 0 开始执行程序，并在任务 0 中创建任务 1 为准备执行。任务 1 则在一开始便使用 os_wait 函数等待其他任务的信号；而任务 0 则在 count0 计数达到 10 之后，向任务 1 发送信号，并重新置 count0 为 0。任务 1 接收到信号后，便立即将 count1 计数加 1 并输出，而不用等待下一个定时信号产生。

说明：设定优先级及使用抢先任务切换必须在 RTX-51 FULL 中进行，程序中需要相应的头文件 RTX51.h 支持。

20.2.5 RTX-51 的其他特性

RTX-51 实时多任务操作系统，完全不同于一般的单片机 C51 语言程序。RTX-51 有自己独特的概念和特点。

1. 中断

RTX-51 系统可以使用中断，其中断函数以并行方式工作。中断函数可以与 RTX-51 内核通信，并可以将信号或消息发送到 RTX-51 的指定任务中。在 RTX-51 FULL 中，中断一般配置为一个任务。

2. 信息传递

RTX-51 FULL 支持任务之间的信息交换，可以使用 isr_rcv_message、isr_send_message、

os_send_message 和 os_wait 函数来实现。在 RTX-51 系统中，信息是一个可以被存储器看做是数字或者指针的 16 位数值。RTX-51 FULL 中还可以支持使用存储器库系统的变量信息。

3. CAN 通信

RTX-51 FULL 中集成了一个 CAN 总线通信模块 RTX-51/CAN。通过 RTX-51/CAN 可以轻松实现 CAN 总线的通信。RTX-51 CAN 作为一个任务来使用，可以通过 CAN 网络来实现信息的传递，其他的 CAN 终端可以配置为一般的 C51 语言程序，也可以是 RTX-51 的实时操作系统。

4. BITBUS 通信

RTX-51 FULL 系统中集成了 BITBUS 主控制器和从控制器。BITBUS 任务主要用于支持与 Intel 8044 之间的信息传递。

5. 事件

在 RTX-51 多任务操作系统中，os_wait 函数所支持的事件有如下几种。

- Timeout: 在所运行的任务中暂停一定数量的时钟周期。
- Interval: Interval 和 Timeout 相似，Interval 主要用于必须同步执行的任务。
- Signals: 用于 RTX-51 各个任务之间的协调。
- Messages: 用于 RTX-51 各个任务之间的信息交换，其只在 RTX-51 FULL 中使用。
- Interrupt: 在 RTX-51 中用于等待 8051 硬件中断的任务，其只在 RTX-51 FULL 中使用。
- Semaphores: 用于共享系统资源的管理，其只在 RTX-51 FULL 中使用。

20.3 RTX-51 Tiny 的系统函数

RTX-51 Tiny 是 RTX-51 FULL 的一个简化版本，其可以完成绝大多数的实时多任务操作。下面首先介绍 RTX-51 Tiny 的系统函数。这些函数主要用于任务管理、任务通信及其他服务。RTX-51 Tiny 的系统函数可以直接被 C51 语言调用。

在 RTX-51 Tiny 的系统函数中，以“os_”开头的函数可以被任务专用，而以“isr_”开头的函数则表示可以被 C51 语言的中断函数专用。

在使用 RTX-51 Tiny 的系统函数时，需要在程序中加入 RTX51TNY.h 头文件。在该头文件中，提供了 RTX-51 Tiny 系统函数的说明及所有常数声明。

20.3.1 发送信号函数 isr_send_signal

发送信号函数 isr_send_signal 主要用于向一个任务发送信号。其函数原型如下：

```
char isr_send_signal(unsigned char taskid);
```

其中，参数 taskid 表示接收信号的任务号。发送信号函数 isr_send_signal 如果返回 0，则表示信号发送成功，如果返回-1，则表示指向的任务不存在。

发送信号函数 isr_send_signal 在向 taskid 所指定的任务发送信号时，如果该任务正在等待信号，则信号到达后，任务再次执行。如果任务正在执行其他操作，则信号将被存储在所访问的任务信号标志中。使用函数 isr_send_signal 的程序示例如下：

```
#include <RTX51TNY.h>

void task_isrsendsignal(void) interrupt 2
{
    .....
    isr_send_signal(3);           //向任务 3 发送信号
}
```

```
.....  
}
```



说明：发送信号函数 `isr_send_signal` 只能被中断函数所调用。

20.3.2 清除信号标志函数 `os_clear_signal`

清除信号标志函数 `os_clear_signal` 主要用于清除指定任务的信号标志。其函数原型如下：

```
char os_clear_signal (unsigned char taskid);
```

其中，参数 `taskid` 表示所需要清除信号标志的任务号。清除信号标志函数 `os_clear_signal` 如果返回 0，则表示信号标志清除成功，如果返回-1，则表示指向的任务不存在。

清除信号标志函数 `os_clear_signal` 清除指定任务的信号标志，主要用于选择所定义的输出状态。使用函数 `os_clear_signal` 的程序示例如下：

```
#include <RTX51TNY.h>  
  
void task_osclearsignal (void) _task_ 2  
{  
.....  
os_clear_signal (3);          //清除任务 3 中的信号标志  
.....  
}
```

20.3.3 启动任务函数 `os_create_task`

启动任务函数 `os_create_task` 主要用于启动指定任务号的任務。其函数原型如下：

```
char os_create_task (unsigned char taskid);
```

其中，参数 `taskid` 表示所需要启动任务的任务号，`taskid` 必须与任务描述的数字相一致，可取值的范围为 0~15。启动任务函数 `os_create_task` 如果返回 0，则表示启动任务成功，如果返回-1，则表示指向的任务不存在或任务无法启动。

启动任务函数 `os_create_task` 启动指定任务标号的任务，将该任务标记为 **READY** 状态。此后，RTX-51Tiny 按照内核的调用规则来执行任务。使用函数 `os_create_task` 的程序示例如下：

```
#include <RTX51TNY.h>  
#include <stdio.h>  
  
void newtask(void) _task_ 3      //任务 3  
{  
.....  
}  
void task_oscreatetask (void) _task_ 2  
{  
.....  
If(os_create_task (3))          //启动任务 3  
{  
    Printf("不能启动任务 3");  
}  
.....  
}
```

20.3.4 删除任务函数 `os_delete_task`

删除任务函数 `os_delete_task` 主要用于删除指定任务号的任務。其函数原型如下：


```
char os_delete_task (unsigned char taskid);
```

其中，参数 `taskid` 表示所需要删除任务的任务号，`taskid` 必须与任务描述的数字相一致，可取值的范围为 0~15。删除任务函数 `os_delete_task` 如果返回 0，则表示删除任务成功，如果返回-1，则表示指向的任务不存在或任务没有启动。

删除任务函数 `os_delete_task` 删除指定任务标号的任务，将该任务从 RTX-51 Tiny 系统的任务列表中删除。使用函数 `os_delete_task` 的程序示例如下：

```
#include <RTX51TNY.h>
#include <stdio.h>

void task_osdeletetask (void) _task_ 2
{
.....
If(os_delete_task (3))          //删除任务 3
{
    Printf("无法删除任务 3");
}
.....
}
```

 **说明：**只有原来用 `os_create_task` 函数说明的任务才能删除。对于正在运行中的任务，也可以使用该函数停止并删除。

20.3.5 当前任务号函数 `os_running_task_id`

当前任务号函数 `os_running_task_id` 主要用于获得当前运行任务的任务号。其函数原型如下：

```
char os_running_task_id (void);
```

其中，当前任务号函数 `os_running_task_id` 的返回值表示当前任务的任务号。使用函数 `os_delete_task` 的程序示例如下：

```
#include <RTX51TNY.h>
#include <stdio.h>

void task_osrunningtaskid (void) _task_ 2
{
unsigned char rid;
rid= os_running_task_id();
printf("rid=%d\n",rid);
.....
}
```

该程序运行时，将输出当前运行的任务号，这里任务号是 2。

20.3.6 发送信号函数 `os_send_signal`

发送信号函数 `os_send_signal` 主要用于向一个任务发送信号。其函数原型如下：

```
char os_send_signal(unsigned char taskid);
```


其中，参数 `taskid` 表示接收信号的任务号。发送信号函数 `os_send_signal` 如果返回 0，则表示信号发送成功，如果返回-1，则表示指向的任务不存在。

发送信号函数 `os_send_signal` 在向 `taskid` 所指定的任务发送信号时，如果该任务正在等待信号，则信号到达后，任务再次执行。如果任务正在执行其他操作，则信号将被存储在所访问的任务信号标志中。使用函数 `os_send_signal` 的程序示例如下：

```
#include <RTX51TNY.h>

void task_ossendsignal(void) _task_ 2
{
```

```
.....  
os_send_signal(3);          //向任务3发送信号  
.....  
}
```

说明：发送信号函数 `os_send_signal` 不同于 `isr_send_signal` 函数，其不能被中断函数所调用。

20.3.7 等待函数 `os_wait`

等待函数 `os_wait` 主要用于暂停当前任务，等待一个或多个事件发生。其函数原型如下：
`char os_wait(unsigned char event_sel, unsigned char ticks, unsigned int dummy);`
其中，参数 `event_sel` 表示等待发生的事件。可以选择的事件有如下几种形式。

- `K_IVL`：等待的时间间隔。
- `K_SIG`：等待的信号。
- `K_TMO`：超时，即等待的时间到。

这些事件可以单独使用，也可以在一起组合使用，示例如下：

```
event_sel=K_TMO | K_IVL;  
event_sel=K_SIG | K_IVL;
```

参数 `ticks` 表征了需要等待完成的时间到（`K_TMO`）或时间间隔（`K_IVL`）所用的计时数。如果没有使用事件 `K_IVL` 或 `K_TMO`，则该参数无意义。

参数 `dummy` 在 `RTX-51` 中不用，其主要用于对 `RTX-51` 的兼容性。

等待函数 `os_wait`，如果返回 `SIG_EVENT`，则表示信号被成功接受；如果返回 `TMO_EVENT`，则表示时间到事件 `K_TMO` 发生或时间间隔 `K_IVL` 完成。如果返回 `NOT_OK`，则表示该函数中所设置的 `event_sel` 参数无效。

在 `RTX-51 Tiny` 中，等待函数 `os_wait` 暂停当前任务的执行，等待一个或多个事件的发生后再继续执行。这里的事件可以是时间间隔、时间到或来自其他任务以及中断的信号，由 `event_sel` 参数指定。使用函数 `os_wait` 的程序示例如下：

```
#include <RTX51TNY.h>  
  
void task_oswait(void)      _task_ 2  
{  
.....  
while(1)  
{  
    char osevent;  
    osevent=os_wait(K_SIG+K_TMO,50,0);  
    switch(osevent)  
    {  
        case TMO_event:    //时间到  
            .....        //任务处理  
            break;  
        case SIG_event:    //信号到  
            .....        //任务处理  
            break;  
        default:  
            break;  
    }  
}  
.....  
}
```


20.3.8 等待函数 os_wait1

等待函数 `os_wait1` 主要用于暂停当前任务，等待信号的到来。其函数原型如下：

```
char os_wait1(unsigned char event_sel);
```

其中，参数 `event_sel` 表示等待发生的事件，其不同于 `os_wait` 函数，只能设置为 `K_SIG`。

等待函数 `os_wait1` 如果返回 `SIG_EVENT`，则表示信号被成功接受；如果返回 `NOT_OK`，则表示该函数中所设置的 `event_sel` 参数无效。

在 RTX-51 Tiny 中，等待函数 `os_wait1` 是 `os_wait` 函数的子集。`os_wait1` 函数暂停当前任务的执行，等待一个事件发生后再继续执行。使用函数 `os_wait1t` 的程序示例如下：

```
#include <RTX51TINY.h>

void task_oswait1(void)    _task_ 2
{
    .....
    while(1)
    {
        char osevent;
        osevent=os_wait1(K_SIG);
        switch(osevent)
        {
            case SIG_event:    //信号到
                .....        //任务处理
                break;
            default:
                break;
        }
    }
    .....
}
```

20.3.9 等待函数 os_wait2

等待函数 `os_wait2` 主要用于暂停当前任务，等待一个或多个事件发生。其函数原型如下：

```
char os_wait2(unsigned char event_sel, unsigned char ticks);
```

其中，参数 `event_sel` 表示等待发生的事件。可以选择的事件有如下几种形式。

- `K_IVL`：等待的时间间隔。
- `K_SIG`：等待的信号。
- `K_TMO`：等待的时间到。

这些事件可以单独使用，也可以在一起组合使用，示例如下：

```
event_sel=K_TMO | K_IVL;
event_sel=K_SIG | K_IVL;
```


参数 `ticks` 表征了需要等待完成的时间到 (`K_TMO`) 或时间间隔 (`K_IVL`) 所用的计时数。如果没有使用事件 `K_IVL` 或 `K_TMO` 则该参数无意义。

等待函数 `os_wait2` 如果返回 `SIG_EVENT`，则表示信号被成功接受；如果返回 `TMO_EVENT`，则表示时间到事件 `K_TMO` 发生或时间间隔 `K_IVL` 完成。如果返回 `NOT_OK`，则表示该函数中所设置的 `event_sel` 参数无效。

在 RTX-51 Tiny 中，等待函数 `os_wait2` 与 `os_wait` 函数的区别在于少了一个 `dummy` 参数。`os_wait2` 函数暂停当前任务的执行，等待一个或多个事件的发生后再继续执行。这里的事件可以是时间间隔、时间到或来自其他任务及中断的信号，由 `event_sel` 参数指定。使用函数 `os_wait2` 的程序示例如下：

```
#include <RTX51TINY.h>

void task_oswait2(void)    _task_ 2
{
.....
while(1)
{
    char osevent;
    osevent=os_wait2(K_SIG+K_TMO,50,0);
    switch(osevent)
    {
        case TMO_event:    //时间到
            .....        //任务处理
            break;
        case SIG_event:    //信号到
            .....        //任务处理
            break;
        default:
            break;
    }
}
.....
}
```

说明：os_wait1 和 os_wait2 函数主要用于一些对代码体积要求比较严格的程序。

20.4 RTX-51 Tiny 的任务管理

RTX-51 Tiny 作为 RTX-51 FULL 的一个子集，其任务管理继承了 RTX-51 FULL 的特性，又有自身的特点。下面首先介绍 RTX-51 Tiny 的任务状态，然后介绍所支持的事件及任务切换。

20.4.1 RTX-51 Tiny 的任务状态

RTX-51 Tiny 中支持 5 种任务状态，任何一个任务必须处于其中一个确定的状态。这 5 种状态分别如下所述。

- **READY**：任务正在等待运行。其前面的任务完成后，按照 RTX-51 Tiny 的规则开始运行处于等待状态的任务。
- **RUNING**：任务正在运行。RTX-51 Tiny 中规定，每一个时刻只能有一个任务处于正在运行状态。
- **WAITING**：任务正在处于等待状态。当指定的事件发生时，任务进入 READY 状态。
- **DELETED**：任务被删除，不开始执行。
- **TIMEOUT**：任务被一个循环超时所中断，该状态与状态 READY 相似。所不同的是，这是由 RTX-51 Tiny 操作系统的内部循环任务切换操作而产生的。

20.4.2 RTX-51 Tiny 的事件

RTX-51 Tiny 中支持的事件包括信号 SIGNAL、超时 TIMEOUT 和时间间隔 INTERVAL。这些事件在 os_wait 函数中使用，下面分别介绍。

- **信号 SIGNAL**：主要用于多任务之间通信的位。在 RTX-51 Tiny 系统中，使用系统函数置位或清除信号 SIGNAL。一个任务在运行前可以使用 os_wait 函数来等待信号 SIGNAL 置位，如果信号 SIGNAL 未置位，则该任务将不执行，直到信号 SIGNAL 置位后才返回到 READY 状态，并可以被 RTX-51 Tiny 再次执行。

- 超时 TIMEOUT: 主要用于 os_wait 函数开始的时间延时, 由定时器脉冲来确定持续时间。如果某个任务调用 os_wait 函数带有 TIMEOUT 参数, 则其将被挂起直到延时结束, 接着将返回到 READY 状态并可以被再次运行。
- 时间间隔事件 INTERVAL: 主要用于 os_wait 函数开始的时间间隔, 由定时器脉冲来确定间隔延时时间。这里需要指出的是, 由于 RTX-51 的定时器是不复位的, 因此事件 INTERVAL 将与一直处于运行状态的定时器一起工作, 这与 TIMEOUT 不同。如果某个任务是在同步间隔内执行的, 则其可以使用时间间隔事件 INTERVAL。

20.4.3 RTX-51 Tiny 的任务切换

RTX-51 Tiny 的循环任务切换允许“准平行”地执行多个循环或任务。所谓“准平行”, 即各个任务并不是一直并行运行的, 而是各自在一段预定的时间内运行。8051 的 CPU 执行时间被 RTX-51 Tiny 划分为多个时间段, 并为每个任务分配一个时间段。一个任务只能在其分配的时间段内执行, 然后由 RTX-51 Tiny 内核切换到另一个任务执行。各个时间段的持续时间可以在系统的配置文件 CONF_TNY.A51 中设置, 对应的变量为 TIMESHARING。

另外, 也可以通过 os_wait 函数来通知 RTX-51 Tiny 允许另外一个任务来执行, 这样便不用等待一个任务的时间段结束。os_wait 函数主要是将当前执行的任务挂起, 以等待一个特定的事件发生, 在等待过程中, 便可以运行任何其他任务。

20.5 RTX-51 Tiny 的配置文件

在一般情况下, RTX-51 Tiny 系统按照默认的参数设置来运行。但是其系统定时器间隔及 TIMEOUT 值等都可以重新配置。RTX-51 Tiny 的参数配置文件为 CONF_TNY.A51, 在该配置文件中, 用户可以进行自由配置参数有如下几个。

- RTX-51 Tiny 所使用的系统时钟中断的寄存器组。
- RTX-51 Tiny 的系统定时器间隔。
- RTX-51 Tiny 用于循环切换的 TIMEOUT 值。
- RTX-51 Tiny 系统内部数据存储器的大小。
- RTX-51 Tiny 系统启动后自由堆栈的大小。

配置文件 CONF_TNY.A51 分为两部分, 前半部分主要是 RTX-51 Tiny 的参数配置, 用户可以根据需要更改, 后半部分则不需要更改。下面给出 RTX-51 Tiny 参数配置的部分代码。

```
$NOMOD51 DEBUG
;-----
; 本文件是 RTX-51 Tiny 实时操作系统的一部分
;-----
; CONF_TNY.A51: 这段代码可以用于对 RTX-51 Tiny 实时操作系统的参数进行配置
;
; 复制该文件到用户的工程文件夹下, 然后添加到µVision 项目中便可以进行参数配置
;
; 如果使用命令行工具, 则需要用以下命令来编译本文件
;
; Ax51 CONF_TNY.A51
;
; 如果使用命令行工具, 则需要连接修改过的 CONF_TNY.OBJ 文件到项目中, 使用如下命令
;
; Lx51 <your object file list>, CONF_TNY.OBJ <controls>
;-----
;
```

51 单片机开发与应用技术详解

```
; RTX-51 TINY 的硬件定时器
; =====
;
; 用下面的 EQU 语句初始化 RTX-51 Tiny 实时操作系统后
; 硬件定时器便被定义，RTX-51 TINY 实时操作系统使用 8051 的定时器 0 来控制软件定时器
;
; 定义 RTX-51 Tiny 定时器中断所使用的寄存器组
INT_REGBANK EQU 1 ; 默认为寄存器组 1
;
; 采用 8051 的机器周期来定义硬件定时器溢出
INT_CLOCK EQU 10000 ; 默认为 10000 个机器周期
;
; 使用硬件定时器脉冲来定义循环切换时间数
TIMESHARING EQU 5 ; 默认为 5 个硬件定时器脉冲
; ; 如果设置为 0，将禁止循环任务切换
;
; 用户中断线程：如果应用程序包含用户中断程序则设置为 1
; 这样会比硬件定时器消耗更多的间隔时间
LONG_USR_INTREQ 0 ; 0 用户中断执行速度快
; ; 1 用户中断执行时间长
;
; -----
;
; 8051 硬件定时器中断的用户代码
; =====
;
; 下面的宏定义了硬件定时器中断时执行的代码
;
; 定义在硬件定时器中断时执行的指令
HW_TIMER_CODEMACRO
; ; 默认为空的宏
;
; RETI
; ENDM
;
; -----
;
; 代码限制支持
; =====
;
; 下面的 EQU 语句控制 RTX-51 Tiny 的代码限制
;
; 使能或禁止代码限制
CODE_BANKING EQU 0 ; 默认 0 表示没有代码限制
; ; 1 表示使用代码限制
;
; -----
;
; RTX-51 TINY 的堆栈空间
; =====
;
; 用下面的 EQU 语句来定义堆栈区域所使用的内部 RAM 的大小
```

```

; 以及堆栈区的最小空余空间
; 用一个宏定义了当 CPU 堆栈空间用尽的时候所执行的代码
;
; 定义堆栈空间的最高 RAM 地址
RAMTOP      EQU 0FFH ;默认地址为 (256-1) 字节
;
FREE_STACK  EQU 20      ; 默认堆栈的空余空间为 20 字节
;
;                                ; 0 表示禁用堆栈检查
;
STACK_ERROR MACRO
    CLR EA          ; 禁用中断
    SJMP $          ; 当堆栈空间耗尽时将无限循环
ENDM
;
;
; -----
;
; 8051 CPU 空闲代码
; =====
;
; 很多 8051 单片机支持空闲模式来减少功率消耗以及 EMC
; 下面的宏定义了当没有任务执行时所运行的代码
; 这段代码将 CPU 设置为空闲模式，从而停止指令执行直到 8051 硬件中断发生
;

; 禁止或使能 CPU 空闲模式
CPU_IDLE_CODE EQU 0      ; 0 表示 CPU 空闲模式禁用
; 1 表示 CPU 空闲模式启用

PCON          DATA 087H  ; 8051 单片机中的功率控制寄存器

; 停止 CPU 执行直到硬件中断
CPU_IDLE MACRO
    ORL PCON, #1          ;设置 8051 CPU 为空闲状态
ENDM
;

```

在该文件中包含了多个 RTX-51 Tiny 实时多任务操作系统的设置参数，一般用 EQU 来指定。下面分别介绍这些参数的功能。

- **INT_REGBANK**: 表示 RTX-51 Tiny 的系统中断所采用寄存器组，默认值为寄存器组 1。
- **INT_CLOCK**: 表示 RTX-51 Tiny 的系统时钟间隔。实时操作系统的时钟在所设定的间隔内产生中断，该值规定了 CPU 每次的中断周期。
- **TIMESHARING**: 表示 RTX-51 Tiny 循环任务切换的 TIMEOUT。该数值表示了 8051 CPU 分配给一个任务的定时器脉冲个数，当脉冲完成后便是 TIMEOUT，接着循环切换到另一个任务。如果该值设置为 0，则将禁止循环切换任何。
- **RAMTOP**: 表示 RTX-51 Tiny 所使用的 8051 CPU 内部 RAM 的最高存储地址。一般来说对于 8051 系列，该值应设置为 7FH；而对于 8052 系列，该值应设置为 FFH。
- **FREE_STACK**: 表示 RTX-51 Tiny 以字节为单位规定的自由栈区大小。当进行任务切换的时候，RTX-51 Tiny 在堆栈区内检测所设定的字节数是否可用。如果堆栈区太小，系统将启动宏 STACK_ERROR。该值可设置的范围一般为 0~0FFH，默认为 20。

- **STACK_ERROR**: 表示 RTX-51 Tiny 检测到堆栈区出错时, 所启用的宏。用户可以根据应用程序的需要来改变该宏。

20.6 RTX-51 Tiny 的要求及限定

RTX-51 Tiny 为用户在 8051 系列单片机上运行实时多任务操作系统提供了便利, 然而其对编译环境及硬件有特定的要求。另外, RTX-51 Tiny 中的函数、指针和寄存器的选择与普通的单片机程序也有所区别, 这些在使用时都要及时注意。

20.6.1 使用 RTX-51 Tiny 的要求

使用 RTX-51 Tiny 时, 在编译环境和硬件系统方面的要求如下:

1. 编译环境要求

在使用 RTX-51 Tiny 实时任务操作系统时, 在计算机软件方面, 需要以下支持。

- C51 编译器
- DL51 连接定位器
- A51 宏汇编器

在安装完 μ Vision3 集成开发环境后, RTX-51 Tiny 所需的所有库文件都已经包含。其中库文件 RTX51TINY.LIB 必须位于目录 C51\LIB 下, 头文件 RTX51TINY.H 必须位于目录 C51\INC 下。

2. 硬件系统要求

一个 RTX-51 Tiny 系统的运行, 需要对 8051 单片机进行合理的设置。RTX-51 Tiny 支持多种存储器模式, 存储器模式的选择仅影响应用程序目标文件的定位, 可以在 C51 编译环境中进行设置。RTX-51 Tiny 中的应用程序可以在外部存储器中存取, 当然也可以在没有外部扩展数据存储器的单片机 8051 系统中运行。

一般来说, RTX-51 Tiny 应用程序应采用 SMALL 编译模式, RTX-51 Tiny 的系统变量和应用程序的堆栈区总是位于 8051 内部数据存储器 (DATA\IDATA) 中。

这里需要注意的是, RTX-51 Tiny 和 RTX-51 Full 的系统运行有如下不同的要求。

- RTX-51 Tiny 仅支持循环任务切换, 不支持任务抢占和优先级切换功能。如果希望在应用程序中采用任务抢占功能, 则必须使用 RTX-51 Full。
- RTX-51 Tiny 不支持代码分组程序, 如果希望在代码分组程序中采用多任务操作系统, 则必须使用 RTX-51 Full。

20.6.2 RTX-51 Tiny 的注意事项

除了上述要求外, 在进行 RTX-51 Tiny 程序设计时, 还需要注意以下一些事项。

1. 任务定义

RTX-51 Tiny 最多允许定义 16 个任务, 每个任务可以完成特殊的功能。RTX-51 Tiny 的任务定义格式如下:

```
void function (void) _task_ num
```

其中, function 是任务的函数名, _task_ 是一个系统关键字, num 是任务编号。每个任务都必须对应一个唯一的任务号, 其取值范围为 0~15。示例如下:

```
void task1 (void) _task_ 1
{
    while (1)
```

```

{
    counter ++;
}
//无限循环
//更新变量 counter 的值

```

注意：RTX-51 Tiny 的任务没有返回值和参数。

2. 寄存器组选择

RTX-51 Tiny 的任务分配使用的是寄存器组 0，程序中的其他函数只能使用余下的寄存器组。一般来说，按照编译环境的默认设置 REGISTERBANK (0) 来进行编译即可。RTX-51 Tiny 实时操作系统要求在一个附加寄存器组中包含 6 个常设字节，可以在配置文件 CONF_TINY.A51 中进行设置，相应的设置参数为 INT_REGBANK。

3. 中断处理

RTX-51 Tiny 本身不包括任何中断管理，在应用时必须允许使能 EA 位，以便能够触发中断。RTX-51 Tiny 对 8051 中断允许寄存器的操作与其他 8051 应用程序一样。由于 RTX-51 Tiny 使用的是 8051 内部定时器 0，因此不允许禁止定时器 0 中断。RTX-51 Tiny 可以并行处理中断函数。

4. 再入函数

再入函数在使用时，将它们的参数和局部数据存放在“再入栈”中，这样这些数据可以在重复调用中得到保护。但是，RTX-51 Tiny 不包括任何对“再入栈”的管理，因此在应用程序中使用再入函数时，必须保证这些函数不调用任何 RTX-51 Tiny 系统函数，并且这些再入函数不能被 RTX-51 Tiny 的循环任务切换所中断。

非再入的 C51 语言函数被几个任务或中断服务程序所调用时，由于它们的参数和局部数据存放在静态存储器段中，因此这些数据将会发生覆盖。非再入函数只有在用户程序保证不会发生重复调用的情况下，才可以被几个任务所调用。因此在使用 RTX-51 Tiny 时必须禁止循环任务的切换，并且这些函数还不允许调用任何 RTX-51 Tiny 系统函数。

RTX-51 Tiny 中使用寄存器传递参数和自动变量的 C 语言函数是可再入的，这些函数可以被不同的任务所调用而没有任何限制。

5. 库函数的使用

库函数在使用时必须明确其是否可再入。对于可再入的函数都可以不加限制地应用于所有任务中。对于使用那些非再入函数，用户必须保证它们不能同时被多个任务所调用。

6. 多数据指针和算术处理器

RTX-51 Tiny 实时多任务操作系统不能管理多数据指针和算术处理器这些与硬件有关的操作。但是，如果能够确保 RTX-51 Tiny 在执行与这些硬件有关的操作时没有循环任务切换发生，则用户程序可以使用多数据指针及算术处理器。

20.7 RTX-51 FULL 的系统函数及技术参数

在 RTX-51 FULL 中提供了对实时多任务操作系统更多的功能支持，下面主要介绍一下 RTX-51 FULL 中的系统函数以及相关的技术参数。

20.7.1 RTX-51 FULL 函数一览

RTX-51 Tiny 中的系统函数在 RTX-51 FULL 中同样有，同时，RTX-51 FULL 中还拥有一

些特有的函数。常用的 RTX-51 FULL 的函数，如表 20-1 所示。

表 20-1 RTX-51 FULL 的函数

函 数	描 述	CPU 周期
isr_send_signal	将一个信号发送给任务（从中断调用）	46
os_clear_signal	删除一个已发送的信号	57
os_create_task	将一个任务加入执行队列	302
os_delete_task	从执行队列移除一个任务	172
os_send_signal	将一个信号发送给任务（从任务调用）	408（使用任务切换） 316（使用快速任务切换） 71（不使用任务切换）
os_wait	等待一个事件	68（信号） 160
os_attach_interrupt *	将任务分配给中断源	119
isr_recv_message *	接收一个消息（从中断调用）	71（连同消息）
isr_send_message *	发送一个消息（从中断调用）	53
os_create_pool *	定义一个存储器库	644（大小 20×10B）
os_detach_interrupt *	移除中断分配	96
os_disable_isr *	禁用 8051 硬件中断	81

续表

函 数	描 述	CPU 周期
os_enable_isr *	使能 8051 硬件中断	80
os_free_block *	将一个块送回存储器库	160
os_get_block *	从存储器库取出一个块	148
os_send_message *	发送一个消息（从任务调用）	443（使用任务切换）
os_send_token *	设置一个信号标志（从任务调用）	343（使用快速任务切换） 94（不使用任务切换）
os_set_slice *	设置 RTX-51 系统时钟时间段	67

这里需要注意的是，加“*”的函数只可在 RTX-51 FULL 中使用。另外，RTX-51 FULL 同时还支持调试函数，如表 20-2 所示；以及 CAN 通信函数，如表 20-3 所示。

表 20-2 RTX-51 FULL 的调试和支持函数

函 数	描 述	函 数	描 述
oi_reset_int_mask	禁用 RTX-51 的外部中断源	os_check_semaphore	检查指定信号标志状态的返回信息
oi_set_int_mask	使能 RTX-51 的外部中断源	os_check_semaphores	检查系统中所有信号标志状态的返回信息
os_check_mailbox	检查指定邮箱状态的返回信息	os_check_task	检查指定任务的返回信息
os_check_mailboxes	检查系统中所有邮箱状态的返回信息	os_check_tasks	检查系统中所有任务的返回信息
os_check_pool	检查存储器中块的返回信息		

表 20-3 RTX-51 FULL 的 CAN 通信函数

函 数	描 述	函 数	描 述
-----	-----	-----	-----

can_bind_obj	将一个目标连接到任务；当收到目标时启动任务	can_receive	接收所有目标
can_def_obj	定义 CAN 通信目标	can_request	向指定的目标发送一个远程帧
can_get_status	获得 CAN 控制器状态	can_send	在 CAN 总线上发送一个目标
can_hw_init	初始化 CAN 硬件控制器	can_task_create	创建 CAN 通信任务
can_start	启动 CAN 通信	can_unbind_obj	将目标和任务间的绑定拆分开
can_stop	停止 CAN 通信	can_wait	等待一个绑定目标的接收
can_read	直接读取一个目标的数据	can_write	将新数据写入一个目标但不发送

20.7.2 RTX-51 的技术参数

RTX-51 实时多任务操作系统运行于 8051 硬件平台，其根据 51 系列单片机的特点进行了特定的优化和限定。RTX-51 FULL 和 RTX-51 Tiny 的技术参数要求，如表 20-4 所示。

表 20-4 RTX-51 的技术参数要求

描 述	RTX-51 FULL	RTX-51 Tiny
任务数量	最多 256 个，可同时激活 19 个	最多 16 个
RAM 需求	40 到 46 字节 DATA 空间 20 到 200 字节 IDATA 空间（用户堆栈） 最少 650 字节 XDATA 空间	7 字节 DATA 空间 3 倍于任务数量的 IDATA 空间

续表

描 述	RTX-51 FULL	RTX-51 Tiny
代码大小限制	6KB~8KB	900 字节
硬件资源要求	定时器 0 或定时器 1	定时器 0
系统时钟	1000 到 40000 个周期	1000 到 65535 个周期
中断请求时间	<50 个周期	<20 个周期
任务切换时间	70 到 100 个周期（快速任务） 180 到 700 个周期（标准任务），取决于堆栈的负载	100~700 个周期，取决于堆栈的负载
邮箱系统	8 个分别带有整数入口的信箱	不支持
内存池	最多 16 个内存池	不支持
信号量	8×1 位	不支持

20.8 小结

本章详细介绍了运行于 8051 硬件平台的 RTX-51 实时多任务操作系统。RTX-51 的程序不同于普通的单片机程序，这里对 RTX-51 的任务调度、系统函数、任务管理及 RTX-51 Tiny 的配置进行了详细的阐述。熟练掌握和运用本章内容，可以简化复杂的多任务单片机系统设计。

第 21 章 Keil μ Vision3 中的 单片机硬件资源仿真

8051 单片机内部集成了多种片上资源，包括并行 I/O 端口、定时器/计数器、串行接口和中断系统等。这些丰富的片上资源构成了单片机强大的功能。开发单片机程序需要对单片机的各种片上资源进行操作。如果能够在程序设计的同时，对代码的执行及各种片上资源的响应进行仿真，则可以大大加速开发的进度并提高程序的准确性。

Keil μ Vision3 集成开发环境中提供了对 51 系列的单片机强大的支持，它不仅支持丰富的单片机型号，同时还对各种片上资源提供了仿真支持。本章将主要介绍 Keil μ Vision3 中的单片机硬件资源仿真。

21.1 仿真概述

在 Keil μ Vision3 集成开发环境中，对标准的 8051 单片机及各个公司推出的新型单片机均提供了支持。即使没有外部硬件支持，在 Keil μ Vision3 集成开发环境中也可以完美仿真模拟程序的执行。对于标准的 8051 单片机，其支持的片上资源仿真包括以下几项。

- 并行 I/O 接口
- 定时器/计数器
- 串行接口
- 中断
- 寄存器

对于一些新型的增强型单片机，除了支持标准的 8051 片上资源外，还可以仿真其他一些片上资源。示例如下：

- 看门狗定时器
- A/D 转换器
- D/A 转换器
- 低功耗模式
- Flash/EEPROM 存储器
- I2C 总线
- CAN 总线

单片机的型号不同，其内部集成的资源也有所不同。下面主要介绍 8051 标准的片上资源及一些常用的增强型单片机资源的仿真操作。

21.2 并行 I/O 端口的仿真

典型的 8051 单片机具有 4 个 8 位的并行 I/O 端口，分别为 P0、P1、P2 和 P3，共 32 条 I/O 线。这些 I/O 端口是双向 I/O 端口，每个端口均可以用做输入和输出。对于一些增强型的单片

机,可能包含更多的 I/O 端口,而有些简化的单片机则省略了一些 I/O 端口。这里以典型的 8051 单片机为例,介绍单片机并行 I/O 端口的仿真。

在 8051 单片机中,这些 I/O 端口分别对应 4 个特殊功能寄存器 P0、P1、P2 和 P3。当在 Keil μ Vision3 集成开发环境中建立一个工程项目,并指定程序开发所采用的单片机后,系统自动将该单片机所包含的并行 I/O 端口资源加载。此时,在程序的仿真模式下便可以进行并行 I/O 端口的读写仿真操作。

下面举例说明并行 I/O 端口的仿真操作,具体操作步骤如下:

(1)首先,在 Keil μ Vision3 集成开发环境中,选择“Project”→“New”→“ μ Vision Project”命令,新建一个工程,并保存。

(2)在弹出的“Select Device for Target”对话框中选择 Atmel 公司的 AT89S52,如图 21-1 所示。

(3)单击“确定”按钮,此时弹出“ μ Vision3”对话框,如图 21-2 所示。单击“是”按钮,完成工程的建立。

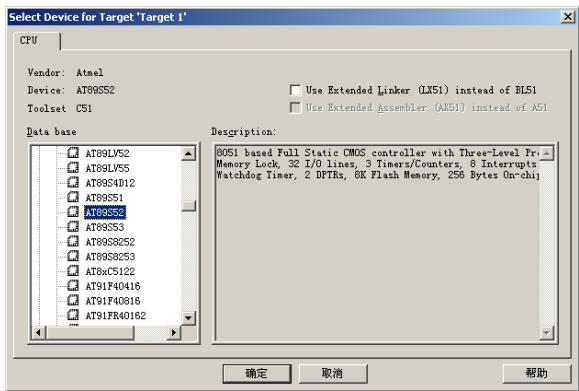


图 21-1 选择单片机 AT89S52

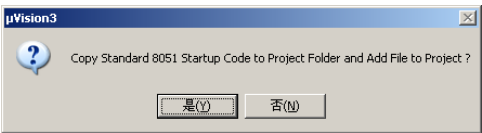


图 21-2 “ μ Vision3”对话框

(4)选择“File”→“New”命令,新建一个程序文件,并保存为*.C 文件,可以在其中输入程序代码。示例如下:

```
#include <reg52.h>                                //头文件

void main()                                         //主函数
{
    int ReadPort;                                  //声明变量
    while(1)                                       //主循环
    {
        ReadPort=P1;                              //读端口 P1
        P2=ReadPort;                              //写端口 P2
    }
}
```

该程序在主循环中,首先读端口 P1 并将端口值保存在变量 ReadPort 中,接着将 P1 端口的值再写入端口 P2。在程序执行时,如果改变 P1 端口的值,P2 端口应该随之改变。

(5)选择“Debug”→“Start/Stop Debug Session”命令,进入程序仿真调试环境。

(6)选择“Peripherals”→“I/O-Ports”→“Port 1”命令,打开并行 I/O 端口 P1 的仿真界面。选择“Peripherals”→“I/O-Ports”→“Port 2”命令,打开并行 I/O 端口 P2 的仿真界面。P1 端口和 P2 端口的仿真界面,如图 21-3 所示。其中 P1、P2 为端口寄存器及其值,Pins 为引脚及其值。

(7)选择“Debug”→“Run”命令,程序便开始仿真执行。此时,改变 P1 端口各个引脚

的值，P2 端口便随之改变，如图 21-4 所示。

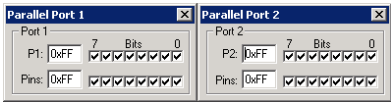


图 21-3 P1 端口和 P2 端口的仿真界面

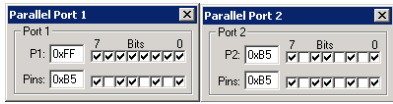


图 21-4 P1 端口和 P2 端口引脚的变化

(8) 当仿真操作完毕后，选择“Debug”→“Stop Running”命令，便可以结束程序的仿真运行。

(9) 选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。

在以上的仿真操作中，也可以选择“Debug”→“Step”命令或“Debug”→“Step Over”命令单步执行程序，来进行程序仿真。

21.3 定时器/计数器的仿真

典型的 8051 单片机具有两个定时器/计数器 T0 和 T1，部分增强型单片机还有定时器/计数器 T2 及其他一些定时器/计数器。下面以定时器/计数器 T0、T1 和 T2 为例，介绍定时器/计数器的仿真。其他类型的定时器/计数器的仿真操作类似。

21.3.1 定时器/计数器 T0 和 T1 的仿真界面

在 Keil μ Vision3 集成开发环境，定时器/计数器 T0 和 T1 的仿真界面，分别如图 21-5 和图 21-6 所示。从图 21-5 和图 21-6 中可以看出，定时器/计数器 T0 和 T1 具有相同的仿真界面，因此其功能也类似。

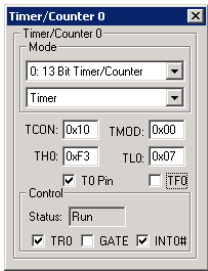


图 21-5 定时器/计数器 T0 的仿真界面

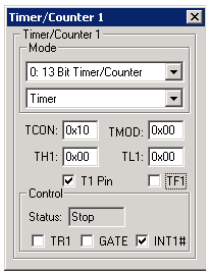


图 21-6 定时器/计数器 T1 的仿真界面

定时器/计数器的仿真界面包含三个区域，可以设置并实时显示该定时器/计数器的工作模式、寄存器值等。下面分别进行介绍。

“Mode”区域用来设置并实时显示定时器/计数器的工作方式。其中第一个下拉列表框中可以选择定时器/计数器的工作模式，包括如下所示的 4 个选项。

- “0:13 Bit Timer/Counter”：定时器/计数器的工作模式 0，13 位计数器。
- “1:16 Bit Timer/Counter”：定时器/计数器的工作模式 1，16 位计数器。
- “2:8 Bit auto-reload”：定时器/计数器的工作模式 2，自动重新装入的 8 位计数器。
- “3:Two 8 Bit Timer/Cnt”：定时器/计数器 T0 的工作模式 3，两个相互独立的 8 位计数器。

对于定时器/计数器 T1，则此项为“3:Disabled”，表示该工作模式无效。

“Mode”区域的第二个下拉列表框，用来选择定时方式还是计数方式，包括如下所示的两个选项。

- “Timer”：表示该定时器/计数器工作于定时方式。
- “Counter”：表示该定时器/计数器工作于计数方式。

寄存器区域用来设置并实时显示相关的寄存器值，包括如下所示的几项。

- “TCON”：用于设置并实时显示定时器/计数器的中断控制寄存器 TCON 的值。
- “TMOD”：用于设置并实时显示定时器/计数器的方式控制寄存器 TMOD 的值。

- “TH0”或“TH1”：用于设置并实时显示定时器/计数器 TH0 或 TH1 的值。
- “TL0”或“TL1”：用于设置并实时显示定时器/计数器 TL0 或 TL1 的值。
- “T0 Pin”或“T1 Pin”：用于计数方式时，计数脉冲的外部输入引脚 P3.4 (T0) 或 P3.5 (T1)。
- “TF0”或“TF1”：定时器/计数器 T0 和 T1 的溢出标志位。

“Control”区域用于控制定时器/计数器的运行，包括如下所示的几项。

- “Status”：显示该定时器/计数器的运行状态。如果显示为“Run”，表示该定时器/计数器正在运行；如果显示为“Stop”，表示该定时器/计数器没有运行。
- “TR0”或“TR1”：定时器/计数器 T0 和 T1 的启/停控制位 TR1/TR0。
- “GATE”：定时器/计数器的门控制位。
- “INT 0#”或“INT 1#”：定时器/计数器相关的 $\overline{\text{INT0}}$ 引脚和 $\overline{\text{INT1}}$ 引脚。

在程序仿真执行的时候，可以在定时器/计数器的仿真界面上，实时观察工作状态及各个寄存器的值。同时，也可以手动更改各个值，来测试程序的运行情况。

21.3.2 定时器/计数器 T2 的仿真界面

对于常用的 52 子系列单片机，其除了包含定时器/计数器 T0 和 T1 外，还扩展了定时器/计数器 T2。在 Keil μ Vision3 集成开发环境，定时器/计数器 T2 的仿真界面，如图 21-7 所示。

定时器/计数器 T2 的仿真界面包含三个区域，可以设置并实时显示定时器/计数器 T2 各个寄存器值等。下面分别进行介绍。

寄存器区域用来设置并实时显示相关寄存器的值，包括如下所示的几项。

- “Mode”：用来实时显示定时器/计数器 T2 的工作方式，显示内容随寄存器的设置而不同。
- “T2CON”：用来设置定时器/计数器 T2 的控制寄存器。
- “T2”：用来设置并显示定时器/计数器 T2 的值。
- “RCAP2”：用来设置并显示捕获模式下的定时器/计数器值。
- “TR2”：定时器/计数器 T2 的溢出中断请求标志位。
- “C/T2#”：定时器/计数器 T2 的定时或计数功能选择位。
- “CP/RL2#”：定时器/计数器 T2 的捕获或重新再装入选择位。
- “EXEN2”：定时器/计数器 T2 的外部触发允许标志位。
- “TCLK”：定时器/计数器 T2 的串行口发送时钟标志位。
- “RCLK”：定时器/计数器 T2 的串行口接收时钟标志位。

“I/O”区域用于设置并实时显示定时器/计数器 T2 相关的 I/O 引脚，包括如下所示的几项。

- “T2EX”：对应 P1.1 引脚，不同模式下有不同的含义。
- “T2 Pin”：对应 P1.0 引脚，不同模式下有不同的含义。

“IRQ”区域用于设置并实时显示定时器/计数器 T2 的中断，包括如下所示的几项。

- “TF2”：定时器/计数器 T2 的溢出中断请求标志位。
- “EXF2”：定时器/计数器 T2 的外部中断请求标志位。

在程序仿真执行的时候，可以在定时器/计数器 T2 的仿真界面上，实时观察工作状态及各个寄存器的值。同时，也可以手动更改各个值，来测试程序的运行情况。

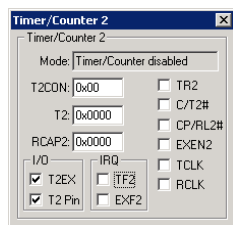


图 21-7 定时器/计数器 T2 的仿真界面

21.3.3 定时器/计数器的仿真操作

这里以定时器/计数器 T0 的工作模式 0 为例，介绍定时器/计数器的仿真操作。假设采用 AT89C51 单片机，外接 6MHz 晶振，采用定时器 T0 的模式 0 产生 1ms 的定时，并在 P1.1 端口输出周期为 2ms 的方波。根据前面章节的介绍，可知定时器 T0 的初值应该设置为 TH0=0F0H，TL0=0CH。程序示例如下：

51 单片机开发与应用技术详解

```
#include <reg51.h>                                //头文件

sbit b=P1^1;                                       //位定义

void T0ISR(void) interrupt 1                       //定时器 T0 中断响应
{
    TL0=0x0C;                                     //重置计数初值
    TH0=0x0F0;
    b=~b;                                         //反向
}

void main(void)                                    //主函数
{
    b=0;                                          //初始化 P1^1=0
    TMOD=0x00;                                   //设置定时器 T0 为模式 0
    TL0=0x0C;                                    //初始化
    TH0=0x0F0;
    TR0=1;                                       //启动 T0
    ET0=1;
    EA=1;                                       //开中断
    while(1)                                    //主循环
    {
    }
}
```

在上面的程序中，将位变量 **b** 指向 **P1^1**。在主程序中初始化定时器 **T0**，接着开相应的中断，并进入主循环。主循环中不进行任何操作。定时器溢出时将触发中断，在中断函数 **T0ISR** 中首先重置计数初值，接着将 **P1^1** 反向。

下面介绍定时器/计数器的仿真操作，具体操作步骤如下：

- (1) 首先，在 Keil μ Vision3 集成开发环境中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89C51，如图 21-8 所示。
- (3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 21-9 所示。单击“是”按钮，完成工程的建立。
- (4) 选择“Project”→“Options for Target ‘Target 1’”命令，打开“Options for Target ‘Target 1’”对话框，如图 21-10 所示。
- (5) 打开“Target”选项卡，在“Xtal (MHz)”文本框中输入 6.0，表示该单片机外接 6MHz 的晶振作为振荡器。
- (6) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入前面的程序代码。
- (7) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (8) 选择“Peripherals”→“Timer”→“Timer 0”命令，打开定时器/计数器 **T0** 的仿真界面。
- (9) 选择“Peripherals”→“I/O-Ports”→“Port 1”命令，打开并行 I/O 端口 **P1** 的仿真界面。
- (10) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时，定时器/计数器 **T0** 的仿真运行界面如图 21-11 所示，其中显示了 **T0** 为 13 位定时器模式，状态为正在运行，各个寄存器的值随程序的运行而更新。**P1** 端口的仿真运行界面，如图 21-12 所示，从中可以看到 **P1.1** 引脚的电平循环改变。

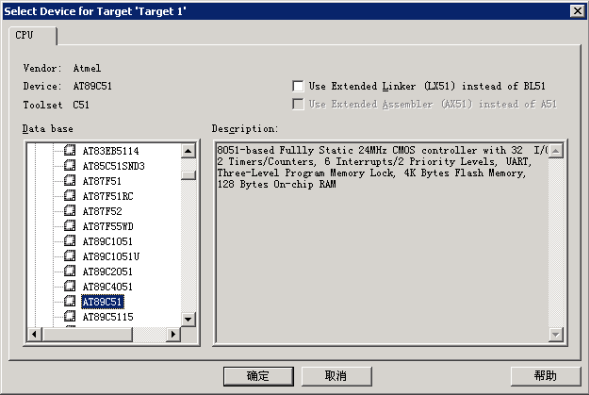


图 21-8 选择单片机 AT89C51

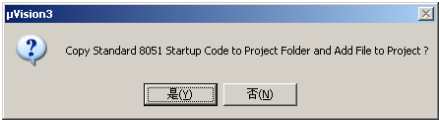


图 21-9 “ μ Vision3”对话框

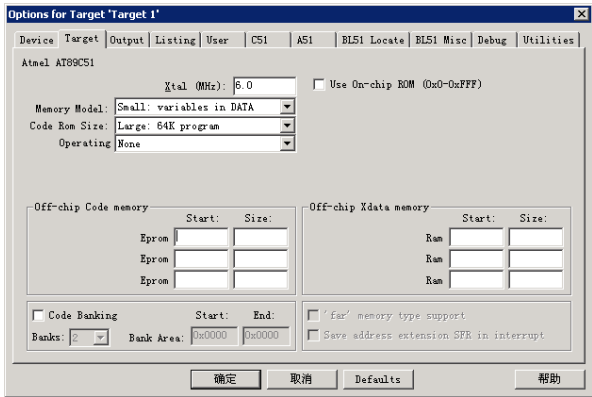


图 21-10 “Options for Target ‘Target 1’”对话框

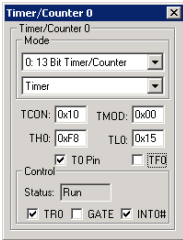


图 21-11 定时器/计数器 T0 的仿真界面



图 21-12 P1 端口的仿真界面

(11) 选择“View”→“Logic Analyzer Window”命令，打开“Logic Analyzer”窗口。

(12) 在“Logic Analyzer”窗口，单击“Setup...”按钮，打开“Setup Logic Analyzer”对话框，如图 21-13 所示。在“Current Logic Analyzer Signals:”文本框中输入变量 b。

(13) 单击“Close”按钮，关闭“Setup Logic Analyzer”对话框。此时，在“Logic Analyzer”窗口，便可以看到 b 变量，即 P1.1 引脚上的波形输出，如图 21-14 所示。从 P1.1 引脚的仿真波形可以看出，该程序通过定时器/计数器 T0 实现了 P1.1 引脚的波形输出。

(14) 当仿真操作完毕后，选择“Debug”→“Stop Running”命令，便可以结束程序的仿真运行。

(15) 选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。

在以上的仿真操作中，也可以选择“Debug”→“Step”命令或“Debug”→“Step Over”命令来单步执行程序，进行程序仿真。对于其他定时器/计数器可采用同样的方法来进行仿真。

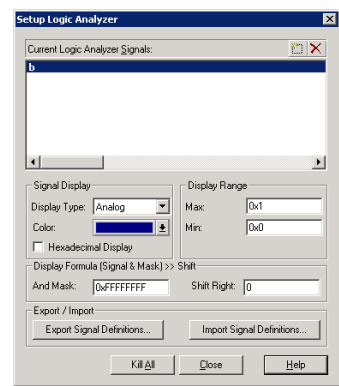


图 21-13 “Setup Logic Analyzer”对话框

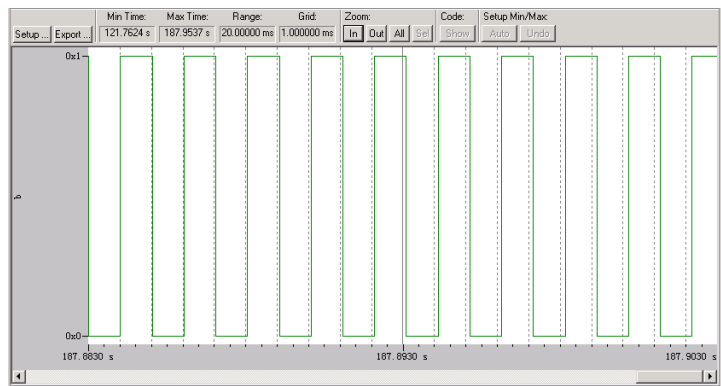


图 21-14 P1.1 引脚的仿真波形

21.4 串行接口的仿真

51 系列单片机提供了功能强大的全双工串行通信接口，部分增强型的单片机会提供多个串行接口。Keil μ Vision3 集成开发环境提供了完整强大的串行接口的仿真环境，可以随时查看设置各个寄存器，也可以仿真字符串数据流的输入输出。

21.4.1 串行接口的仿真界面

在 Keil μ Vision3 集成开发环境，串行接口的仿真界面，如图 21-15 所示。

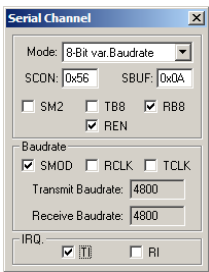


图 21-15 串行接口的仿真界面

串行接口的仿真界面包含三个区域，可以设置并实时显示串行接口的工作模式、寄存器值等。下面分别进行介绍。

寄存器区域的“Mode”用来设置并实时显示串行接口的工作方式。其下拉列表框中可以选择串行接口的工作模式，包括如下所示的 4 个选项。

- “8-Bit Shift Register”：串行口的工作模式 0，即 8 位同步移位寄存器输入/输出方式。
 - “8-Bit var. Baudrate”：串行口的工作模式 1，即 8 位波特率可变的串行异步通信方式。
 - “9-Bit fix Baudrate”：串行口的工作模式 2，即 9 位固定波特率的串行异步通信方式。
 - “9-Bit var. Baudrate”：串行口的工作模式 3，即 9 位可变速异步发送接收方式。
- 寄存器区域的其他项用来设置并实时显示相关的寄存器值，包括如下所示的几项。
- “SCON”：用于选择串行通信的工作方式和某些控制功能，包括串行接口的接收/发送控制及设置状态标志等。
 - “SBUF”：串行接口接收发送的缓冲器。
 - “SM2”：多机通信的控制位。在多机通信模式中，通信的双方分为主机和从机，而 SM2 用于控制从机的接收。
 - “TB8”：在单机通信中，TB8 可以作为奇偶校验位；而在多机串行通信中，可以作为发送地址帧和数据帧的标志位。
 - “RB8”：既可以作为约定的奇偶校验位，也可以是约定的地址/数据标志位。
 - “REN”：接收允许/禁止控制位，相当于串行数据接收的开关。
- “Baudrate”区域用来设置并实时显示串行接口的运行波特率，包括如下所示的几项。
- “SMOD”：波特率倍增位。如果 SMOD=1，则波特率提高 1 倍；如果 SMOD=0，则波

特率不变。

- “RCLK”：串行接口发送时钟标志位。
- “TCLK”：串行接口接收时钟标志位。
- “Transmit Baudrate”：显示串行接口当前的发送波特率。
- “Receive Baudrate”：显示串行接口当前的接收波特率。

“IRQ”区域用于设置并实时显示串行接口的中断，包括如下所示的几项。

- “T1”：发送中断请求标志位，在一帧数据信息发送结束的时候由硬件自动置位，在仿真时也可自己手动置位。
- “R1”：接收中断请求标志位，在接收到一帧的有效数据后由硬件置位。

在程序仿真执行的时候，可以在串行接口的仿真界面上，实时观察工作状态及各个寄存器的值。同时也可以手动更改各个值，来测试程序的运行情况。

21.4.2 串行接口的仿真操作

下面首先介绍使用串行接口寄存器来实现数据发送和接收的仿真操作。串行接口的数据发送需要用到寄存器 SBUF 及 TI 标志位，串行接口的数据接收需要用到寄存器 SBUF 及 RI 标志位。具体的仿真操作步骤如下：

(1) 按照前面的方法，新建一个工程，并选择 Atmel 公司的 AT89S52 单片机。

(2) 选择“Project”→“Options for Target ‘Target 1’”命令，打开“Options for Target ‘Target 1’”对话框。在“Target”选项卡中的“Xtal (MHz)”文本框中输入 11.0592，表示该单片机外接 11.0592MHz 的晶振作为振荡器。如图 21-16 所示。

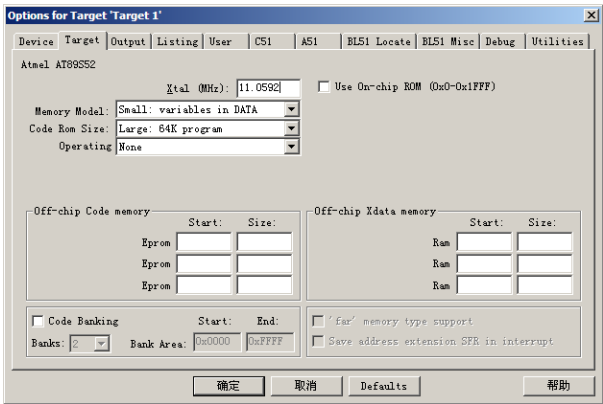


图 21-16 设置晶振频率

(3) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入如下的程序代码。

```
#include <reg52.h> //头文件

voidmain(void) //主函数
{
    int ch;
    SCON=0x50; //初始化串行口模式 1
    TMOD=0x20; //初始化 T1 为定时功能，模式 2
    PCON=0x80; //设置 SMOD=1
    TL1=0xF4; //波特率 4800bit/s，初值
    TH1=0xF4;
    ES=0; //禁止串行中断
```

```
TR1=1; //启动定时器
//串口发送
SBUF=0x76;
while(TI) //等待 TI=1
{
    TI=0; //TI 清零
}
//串口接收
while(RI) //等待 RI=1
{
    ch=SBUF; //读取串行数据
    RI=0; //RI 清零
}
}
```

在该程序中，主函数首先初始化串口为模式 1，然后使用定时器 T1 设置波特率，并且禁止串行中断和启动定时器。接着为 SBUF 赋值，数据将通过串口输出。最后，使用 while 循环来查询 RI。当 RI=1 时，表示串行数据接收完毕，随后将数据读入变量 ch，并清零 RI，以便于接收下一次串行数据。

- (4) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (5) 选择“Peripherals”→“Serial”命令，打开串行接口的仿真界面。
- (6) 按 F10 键开始单步执行程序。当完成串行口及波特率的设置后，在仿真界面中便可以看到串行口的工作模式为“8-Bit var. Baudrate”，即工作模式 1。串行口的发送和接收波特率均为 4800kbit/s。如图 21-17 所示。
- (7) 在串行接口仿真界面中手动置位 TI 和 RI。继续按 F10 键，当执行 SBUF=0x76 语句后，从仿真界面可以看到数值 0x76 已经发送到缓冲区 SBUF。如图 21-18 所示。
- (8) 继续按 F10 键，分别完成串行数据的发送及接收。此时程序中清零 TI 和 RI，并结束串口传输，如图 21-19 所示。
- (9) 语句全部执行完毕后，选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。

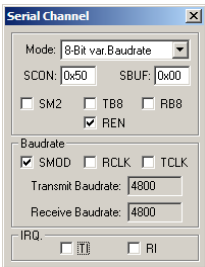


图 21-17 串行接口的工作方式

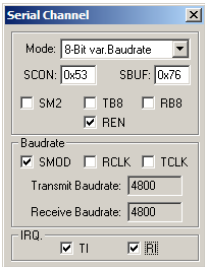


图 21-18 SBUF 赋值并置中断标志位

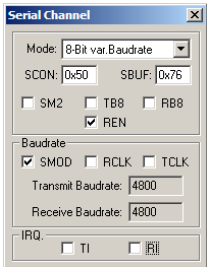


图 21-19 串口传输结束

21.4.3 字符串输入输出的仿真操作

对于一些复杂的程序，需要通过串口发送接收字符串，此时按照前面的方法很难观察串口中的数据流。Keil μ Vision3 集成开发环境中提供了更为强大的串口仿真调试支持。下面介绍具体的字符串输入输出的仿真操作。

- (1) 按照前面的方法，新建一个工程，选择 Atmel 公司的 AT89S52 单片机。同时，在工程中设置单片机外接振荡器的频率为 11.0592MHz。
- (2) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入如下的程序代码。

```
#include<stdio.h>                                //头文件
#include <reg52.h>

voidmain(void)                                    //主函数
{
    char ch;

    SCON=0x50;                                    //串口模式 1，允许接收
    TMOD|=0x20;                                    //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                                    //设置 SMOD=1
    TL1=0xF4;                                       //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                                       //中断
    TR1=1;                                          //启动定时器

    printf("Please input a char!\n");

    while((ch=_getkey ())!=0x0d)                  //获取字符
    {
        printf("The input char=%c, the hex number=%bx\n",ch,ch);    //输出结果
    }
}
```

在该程序中，首先对串口进行了初始化，接着输出一个字符串，然后调用_getkey 函数获取字符，并通过串口 0 输出该字符以及对应的十六进制数。

- (3) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (4) 选择“Peripherals”→“Serial”命令，打开串行接口的仿真界面。
- (5) 选择“View”→“Serial Window”→“UART #0”命令，打开“UART #0”仿真窗口。
- (6) 选择“Debug”→“Run”命令，程序便开始仿真执行。
- (7) 在串行接口的仿真界面上手动置位 TI，此时便可以看到“UART #0”仿真窗口中输出的第一个字符串。如图 21-20 所示。
- (8) 在串行接口的仿真界面上手动清零 TI，然后在“UART #0”仿真窗口输入字符 a。此时，再次在串行接口的仿真界面上手工置位 TI，“UART #0”仿真窗口中输出的第二个字符串。如图 21-21 所示。

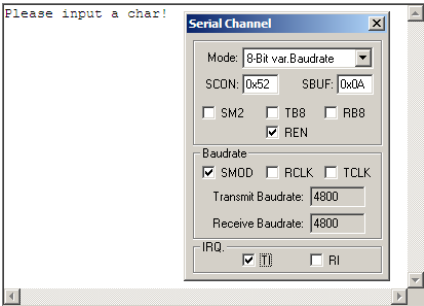


图 21-20 仿真输出第一个字符串

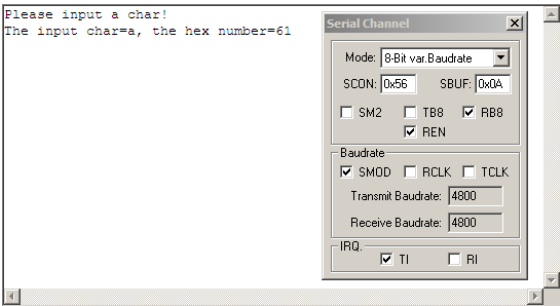


图 21-21 仿真输出第二个字符串

- (9) 当仿真操作完毕后，选择“Debug”→“Stop Running”命令，便可以结束程序的仿真运行。
- (10) 选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。

21.5 中断仿真

51 系列单片机提供有 5 个中断源，包括两个外部中断源、两个定时中断源和一个串行中断源。部分增强型的单片机还提供了更多的中断源。Keil μ Vision3 集成开发环境对每一个型号的单片机均提供了完整的中断仿真支持。

21.5.1 中断系统的仿真界面

下面以 AT89S52 单片机为例，在 Keil μ Vision3 集成开发环境，其中断系统的仿真界面，如图 21-22 所示。

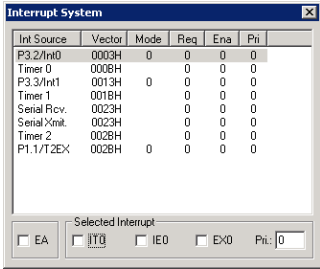


图 21-22 中断系统的仿真界面

中断系统的仿真界面包含三个区域，可以设置并实时显示中断系统的工作状态。下面分别进行介绍。

中断向量表区域用来显示中断向量的使用情况。其中“Int Source”表示中断源，“Vector”表示中断源的入口地址，“Mode”表示中断的触发方式，“Req”表示中断请求标志，“Ena”表示中断请求允许标志，“Pri”表示中断优先级。表中具体各项所表示的含义如下所示。

- “P3.2/Int0”：外部中断 0。
- “Timer0”：定时器/计数器 T0 中断。
- “P3.3/Int1”：外部中断 1。
- “Timer1”：定时器/计数器 T1 中断。
- “Serial Rcv”：串行接口接收中断。
- “Serial Xmit”：串行接口发送中断。
- “Timer2”：定时器/计数器 T2 中断。
- “P1.1/T2EX”：P1.1 引脚的外部中断。

“Selected Interrupt”区域用来设置并实时显示所选择的中断源相关的寄存器值。该区域其中的内容随所选择的中断源不同而有所变化，主要包括如下所示的几项。

- “IT0”或“IT1”：外部中断的中断触发方式控制位。
- “IE0”或“ET1”：外部中断请求标志位。
- “EX0”或“EX1”：外部中断允许/禁止标志位。
- “TF0”或“TF1”：定时器/计数器的溢出标志位。
- “ET0”或“ET1”：定时器/计数器的允许/禁止标志位。
- “RI”：串行接口的接收中断请求标志位。
- “TI”：串行接口的发送中断请求标志位。
- “ES”：串行中断允许/禁止标志位。
- “Pri”：中断的优先级。

“EA”区域用来设置 EA 标志位，“EA”为中断允许或禁止总控制位。当置 EA=0 时，单片机将禁止所有中断，不响应任何中断请求；当置 EA=1 时，单片机允许各个中断，此时还需要由其他标志位确定各个中断的允许或禁止。

在程序仿真执行的时候，可以在中断系统的仿真界面上，实时观察工作状态及各个寄存器的值。同时也可以手动更改各个值，来测试程序的运行情况。

21.5.2 中断系统的仿真操作

单片机的外部中断、定时中断，以及串行中断等，均可以在 Keil μ Vision3 集成开发环境中进行仿真。下面以外部中断源的仿真操作为例进行介绍，具体的仿真操作步骤如下：

- (1) 按照前面的方法，新建一个工程，选择 Atmel 公司的 AT89S52 单片机。
- (2) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输

入如下的程序代码。

```
#include<reg52.h>           //头文件

void ISR0(void) interrupt 0   //外部中断 0 服务例程
{
    P1=P1+1;                 //P1 端口递增
}

void ISR1(void) interrupt 2   //外部中断 1 服务例程
{
    P2=~P2;                  //P2 端口反相
}

void main(void)              //主函数
{
    IP=0x05;                  //外部中断 0 和外部中断 1 设置为高优先级
    IT0=1;                    //外部中断 0 为下降沿触发
    IT1=1;                    //外部中断 1 为下降沿触发
    EX0=1;                    //开 EX0 中断
    EX1=1;                    //开 EX1 中断
    EA=1;                     //开总中断
    P1=0;                     //P1 端口初始化为 0

    while(1)                  //主循环
    {
    }
}
```

在该程序中，分别定义了外部中断 0 和 1 的服务例程。在主函数中，将外部中断 0 和外部中断 1 设置为高优先级，并设置外部中断 0 和 1 为下降沿触发方式。最后打开相应的中断，并置 EA=1 开总中断。在主循环中不进行任何操作。当 $\overline{\text{INT0}}$ （P3.2 引脚）有负跳变时，触发中断服务例程 ISR0，将 P1 端口的数据累加；当 $\overline{\text{INT1}}$ （P3.3 引脚）有负跳变时，将触发中断服务例程 ISR1，置 P2 端口反向。

(3) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。

(4) 选择“Peripherals”→“Interrupt”命令，打开“Interrupt System”中断仿真界面。

(5) 选择“Peripherals”→“I/O-Ports”→“Port 1”命令，打开并行 I/O 端口 P1 的仿真界面。依照同样的步骤，打开并行端口 P2 和 P3 的仿真界面。

(6) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时，从中断仿真界面上可以看到外部中断 0 和外部中断 1 被设置为高优先级，模式为 1 即负跳变触发，并且中断已经使能。外部 I/O 端口 P1 初始化为 0，如图 21-23 所示。

(7) 手动改变 P3.2 引脚的值，此时可以看到中断仿真界面中相应的“Req”项改变，表示该中断触发。此时，P1 端口递增 1。同样，手动改变 P3.3 引脚的值，则触发外部中断 1。中断运行的结果，如图 21-24 所示。

(8) 当仿真操作完毕后，选择“Debug”→“Stop Running”命令，便可以结束程序的仿真运行。

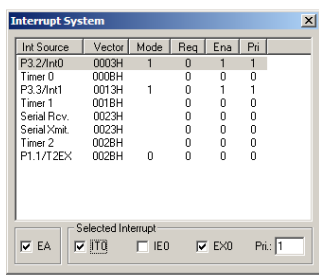


图 21-23 程序仿真运行

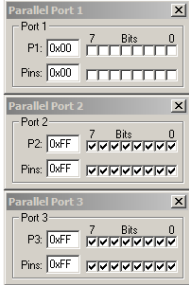


图 21-24 中断运行结果

(9) 选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。

在前面的仿真操作中，除了手动改变外部中断引脚 P3.2 和 P3.3 的值外，还可以在中断仿真界面中，通过改变“IE0”和“IE1”的值来触发外部中断。

对于定时中断和串行中断，可以采用同样的步骤来进行仿真操作，这里不再具体进行介绍。

21.6 看门狗定时器的仿真

目前，大多数单片机都内置有看门狗定时器。当程序正常运行时，看门狗定时器将定时清零。如果遇到系统故障，导致程序跑飞，则看门狗定时器将得不到及时地清零，此时，看门狗定时器溢出，引起系统复位。使用看门狗定时器可以增加软件的可靠性。Keil μ Vision3 集成开发环境对看门狗定时器提供了仿真支持。

21.6.1 看门狗定时器的仿真界面

以 AT89S52 单片机的看门狗定时器为例，在 Keil μ Vision3 集成开发环境中，看门狗定时器的仿真界面如图 21-25 所示。其他类型的单片机中，看门狗定时器的用法类似。

在看门狗定时器的仿真界面中包含如下两项。

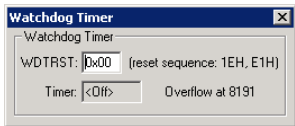


图 21-25 看门狗定时器的仿真界面

显示为“<Off>”，则表示看门狗定时器未启动。

在程序仿真执行的时候，可以在看门狗定时器的仿真界面上，实时观察工作状态及各个寄存器的值。同时也可以手动更改各个值，来测试程序的运行情况。

- “WDTRST”文本框表示看门狗寄存器中的值。一般在程序初始化时，向看门狗寄存器中先写入 0x1E，然后再写入 0xE1，即可启动看门狗定时器。
- “Timer”表示当前看门狗定时器中的计数值，当计数到 8191 时将溢出。因此，应该在溢出之前清零看门狗定时器，否则系统将强制复位。如果该值

21.6.2 看门狗定时器的仿真操作

下面通过实例，介绍 Keil μ Vision3 集成开发环境中看门狗定时器的仿真操作。具体仿真操作步骤如下：

- (1) 按照前面的方法，新建一个工程，选择 Atmel 公司的 AT89S52 单片机。
- (2) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入如下的程序代码。

```
#include<reg52.h>

sfr WDTRST = 0xA6; //声明看门狗定时器

void main() //主函数
```

```
{
int i;
WDTRST=0x1E;           //初始化看门狗定时器
WDTRST=0xE1;

while (1)               //主循环
{
    if (P1==0x01)
    {
        for(i=0;i<100;i++);           //短延时
    }
    else if (P1==0x02)
    {
        for(i=0;i<10000;i++);         //长延时，模拟系统故障
    }
    else
    {
    }

    WDTRST=0x1E;           //喂看门狗定时器
    WDTRST=0xE1;
}
}
```

该程序首先启动看门狗定时器，然后，在 while 主循环中判断 P1 端口的值，当 P1 端口为 0x02 时，执行一个长时间的延时程序，模拟系统故障。此时，看门狗定时器无法在指定的时间内清零，因此，系统将强制复位。

- (3) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (4) 选择“Peripherals”→“Timer”→“Watchdog”命令，打开看门狗定时器仿真界面。
- (5) 选择“Peripherals”→“I/O-Ports”→“Port 1”命令，打开并行 I/O 端口 P1 的仿真界面。
- (6) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时，看门狗定时器开始运行，如图 21-26 所示。
- (7) 当手动置 P1 端口为 0x02 时，由于长时间延时的作用，看门狗定时器无法及时清零，系统将强制复位，如图 21-27 所示。

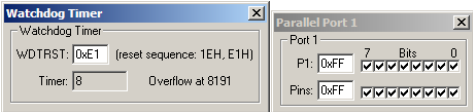


图 21-26 看门狗定时器开始运行

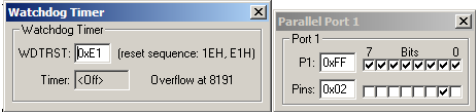


图 21-27 看门狗定时器未能及时清零

- (8) 当仿真操作完毕后，选择“Debug”→“Stop Running”命令，便可以结束程序的仿真运行。
 - (9) 选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。
- 当看门狗定时器溢出时，还可以从 CPU 寄存器界面中看到 CPU 的复位操作。

21.7 A/D 仿真

标准的 51 系列单片机没有集成 A/D 转换的片上资源。随着技术的不断进步，单片机向着高度集成化的方向发展。越来越多的单片机开始集成 A/D 转换器。下面介绍 Keil μ Vision3 集成开发环境中，对单片机片上 A/D 转换器的仿真支持。

21.7.1 A/D 转换器的仿真界面

这里以 Analog Devices 公司的 ADuC 812 单片机为例。该单片机内部集成了一个 8 通道的 A/D 转换器, 转换时间 $5\mu\text{s}$ 、12 位、单电源供电。

在 Keil μ Vision3 集成开发环境中, 其仿真操作界面, 如图 21-28 所示。

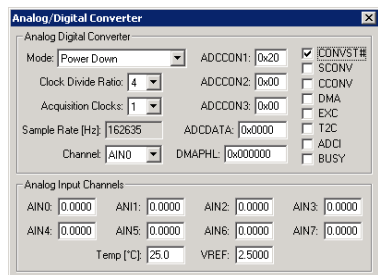
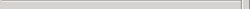


图 21-28 ADuC 812 的 A/D 仿真界面

A/D 转换器的仿真界面包含两个区域,可以设置并实时显示 A/D 转换器工作模式、模拟量及相关的寄存器值等。下面分别进行介绍。

“Analog Digital Converter”区域用于设置并实时显示 A/D 转换器的工作模式及寄存器的值。其中, “Mode”下拉列表框可以设置如下 4 种工作模式。

- “Power Down”: ADC 掉电模式。
- “Normal Mode”: ADC 正常工作模式。

- “Power Down if not Conv.”：如果不执行转换周期，则 ADC 掉电。
 - “Standby if not Conv.”：如果不执行转换周期，则 ADC 待机。
- 另外，“Analog Digital Converter”区域其他相关的设置包括如下所示几项。
- “Clock Divide Ratio”：ADC 时钟分频率，可选择 1、2、4 和 8。
 - “Acquisition Clock”：选择用于输入跟踪/保持放大器采集输入信号的时间，可选择 1、2、3 和 4 四个时钟数。
 - “Sample Rate (Hz)”：显示当前的采样率。
 - “Channel”：模拟采样的通道，可以选择 AIN1～AIN7，以及 Temp。
 - “ADCCON1”：控制转换与采集时间、硬件转换模式及掉电模式的特殊功能寄存器。
 - “ADCCON2”：控制 ADC 通道选择和转换模式的特殊功能寄存器。
 - “ADCCON3”：ADC 转换忙标志指示寄存器。
 - “ADCDATA”：ADC 转换的结果。
 - “CONVST#”：外部引脚 23，用做低电平有效的转换起始输入。
 - “SCONV”：单次转换位。
 - “CCONV”：连续转换位。
 - “DMA”：DMA 模式使能位
 - “EXC”：设置外部触发器使能位。
 - “T2C”：设置定时器 2 转换位。
 - “ADCI”：ADC 中断位。
 - “BUSY”：ADC 忙状态位。
- “Analog Input Channels”区域用于设置并显示各个模拟通道的模拟量，包括如下所示的几项。
- “AIN0”～“AIN7”：模拟通道 0～模拟通道 7 的模拟采集量。
 - “Temp (℃)”：温度模拟量。
 - “VREF”：ADC 的基准电压。

在程序仿真执行的时候,可以在 A/D 转换器的仿真界面上,实时观察工作状态及各个寄存器的值。同时也可以手动更改模拟采集量的值,来测试程序的运行情况。

21.7.2 A/D 转换器的仿真操作

下面通过 ADuC 812 来循环采集 8 个通道的模拟电压数据来讲解 A/D 转换器的仿真操作。具体仿真操作步骤如下：

- (1) 按照前面的方法, 新建一个工程。其中, 选择 Analog Devices 公司的 ADuC 812 单片机, 并设置单片机外接振荡器的频率为 11.0592MHz。
- (2) 选择 “File” → “New” 命令, 新建一个程序文件, 并保存为 *.C 文件, 可以在其中输入如下的程序代码。


```

#include<ADUC812.H>           //头文件
#include<stdio.h>

void main(void)               //主函数
{
    unsigned char chc;        //声明变量，模拟通道数

    SCON=0x50;                //初始化串口
    TMOD|=0x20;
    TL1=0xF4;                 //设置串口波特率为 4800kbps
    TH1=0xF4;
    TR1=1;
    TI=1;

    ADCCON1=0x7C;             //设置控制寄存器

    while(1)                  //主循环连续采集
    {
        unsigned int con_data; //声明变量，A/D 转换的结果
        unsigned char channel; //声明变量，模拟通道数

        chc++;
        chc%=8;               //通道数，0~7
        ADCCON2=( ADCCON2&0xF0)|chc;
        SCONV=1;              //单次转换
        while (ADCCON3&0x80); //读 A/D 值并打印输出结果

        channel=ADCDATAH>>4; //通道数
        con_data=ADCDATAH+((ADCDATAH&0x0F)<<8); //A/D 转换的值

        printf("ADC Channel %bu=0x%4.4X\n",channel, con_data);
    }
}

```

在该程序中，首先初始化串口，然后设置 ADC 采集控制寄存器。接着，在主循环中连续读取各个模拟通道，并进行 A/D 转换，最后输出结果。

(3) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。

(4) 选择“Peripherals”→“A/D Converter”命令，打开 A/D 转换器仿真界面。

(5) 选择“View”→“Serial Window”→“UART #0”命令，打开“UART #0”仿真窗口。

(6) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时，程序连续读取 A/D 转换器 8 个通道的数据并通过串口输出。

(7) 在 A/D 转换器的仿真界面上，改变模拟采集通道的电压值。例如，将 AIN0 设置为 0.0000，AIN1 设置为 0.6250，AIN2 设置为 1.2500，AIN3 设置为 2.5000，其余保持不变。此时从串口输出中便可以看到 A/D 转换器采集后的输出结果，如图 21-29 所示。

该程序将采集到的 A/D 转换数据从串口输出，其输出结果如下：

```

ADC Channel 0=0x0000
ADC Channel 1=0x0400
ADC Channel 2=0x0800
ADC Channel 3=0x0FFF
ADC Channel 4=0x0000
ADC Channel 5=0x0000
ADC Channel 6=0x0000
ADC Channel 7=0x0000

```

由于这里设置的 VREF=2.5000，AIN3 设置为和 VREF 相等，因此第 3 通道输出最大值

0x0FFF; AIN2 设置为 VREF/2, 因此将输出最大值的一半, 即 0x0800; AIN1 设置为 VREF/4, 因此将输出最大值的 1/4, 即 0x0400。

在以上的仿真操作中, 也可以选择“Debug”→“Step”命令或“Debug”→“Step Over”命令来单步执行程序, 来进行程序仿真。

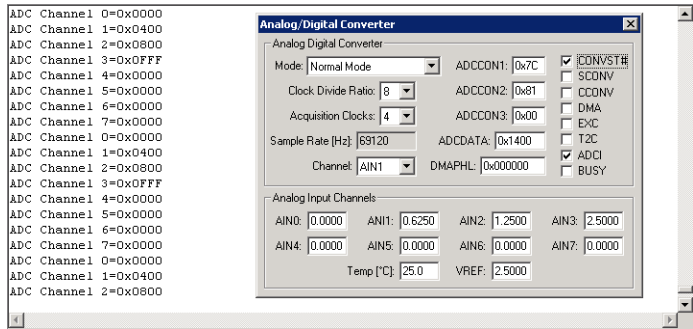


图 21-29 改变模拟电压

21.8 D/A 仿真

标准的 51 系列单片机没有集成 D/A 转换的片上资源。随着技术的不断进步, 单片机向着高度集成化的方向发展。除了 A/D 转换器, 越来越多的单片机也开始集成 D/A 转换器。下面介绍 Keil μ Vision3 集成开发环境中, 对单片机片上 D/A 转换器的仿真支持。

21.8.1 D/A 转换器的仿真界面

这里以 Analog Devices 公司的 ADuC 812 单片机为例。该单片机内部集成了两个 12 位的电压输出型 DAC 转换器。在 Keil μ Vision3 集成开发环境中, 其仿真操作界面, 如图 21-30 所示。

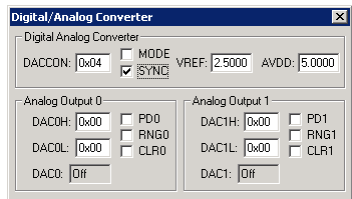


图 21-30 ADuC 812 的 D/A 仿真界面

D/A 转换器的仿真界面包含三个区域, 可以设置并实时显示 D/A 转换器工作模式、输出量及相关的寄存器值等。下面分别进行介绍。

“Digital Analog Converter”区域用于设置并显示 D/A 转换相关的控制以及参考电压等。包括如下所示的几项。

- “DACCON”: D/A 转换器的控制寄存器。
- “MODE”: DAC 转换的模式。
- “SYNC”: 两通道输出同步控制位。
- “VREF”: 片内参考电压。

➤ “AVDD”: 供电电压。

“Analog Output 0”区域用于设置并显示模拟输出通道 0 的数值及相关寄存器等。包括如下所示的几项。

- “DAC0H”: D/A 转换器 0 的高位数据。
- “DAC0L”: D/A 转换器 0 的低位数据。
- “DAC0”: D/A 转换器 0 是否启动。
- “PD0”: 节电模式位。0 表示关闭该通道的 D/A 转换, 1 表示打开该通道的 D/A 转换。
- “RNG0”: 输出范围控制。0 表示输出范围为 0~VREF, 1 表示输出范围为 0~AVDD。
- “CLR0”: 输出清除位。

“Analog Output 1”区域用于设置并显示模拟输出通道 1 的数值及相关寄存器等, 它和“Analog Output 0”区域类似, 这里不再具体介绍。

在程序仿真执行的时候, 可以在 D/A 转换器的仿真界面上, 实时观察工作状态及各个寄存器的值。同时也可以手动更改数值, 来测试程序的运行情况。

21.8.2 D/A 转换器的仿真操作

下面通过 ADuC 812 中集成的两个 D/A 转换器，来输出两个模拟波形。具体仿真操作步骤如下：

(1) 按照前面的方法，新建一个工程。其中，选择 Analog Devices 公司的 ADuC 812 单片机，并设置单片机外接振荡器的频率为 11.0592MHz。

(2) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入如下的程序代码。

```
#include<ADUC812.H>           //头文件
#include<stdio.h>

void main(void)                //主函数
{
    float DAC0V,DAC1V;         //声明变量

    SCON=0x50;                 //初始化串口
    TMOD|=0x20;
    TL1=0xF4;                  //设置串口波特率为 4800kbps
    TH1=0xF4;
    TR1=1;
    TI=1;

    DACCON=0x7B;               //设置 D/A 转换器
    while(1)                   //主循环
    {
        unsigned int i;        //声明变量
                                //循环 2^12=4096

        for(i=0;i<4096;i++)
        {
            DACCON &= ~0x04;    //清零 SYNC 位

            DAC0H=i>>8;         //DAC0
            DAC0L=i&0xFF;
            DAC0V=(float)i*5.0/4095.0; //DAC0 电压值

            DAC1H=(4095- i) >>8; //DAC1
            DAC1L=(4095- i) &0xFF;
            DAC1V=(float) (4095- i) *5.0/4095.0; //DAC1 电压值

            DACCON|=0x04;       //设置 SYNC 位
                                //输出结果
            printf("Output %4.4X to DAC0(%1.4fV)\n", ( unsigned)i,DAC0V);
            printf("Output %4.4X to DAC1(%1.4fV)\n", ( unsigned) (4095- i) ,DAC1V);
        }
    }
}
```

在该程序中，首先初始化串口，然后设置 DAC 为 12-bit 模式，范围是 0-DVD，每循环 4096 次转换 DAC0 和 DAC1。接着，在主循环中连续写 DAC0 和 DAC1，产生两个波形，最后输出结果。

(3) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。

(4) 选择“Peripherals”→“D/A Converter”命令，打开 D/A 转换器仿真界面。

(5) 选择“View”→“Serial Window”→“UART #0”命令，打开“UART #0”仿真窗口。

51 单片机开发与应用技术详解

(6) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时，程序连续向 D/A 转换器 0 和 D/A 转换器 1 写入数据，执行 D/A 转换，并通过串口输出信息。如图 21-31 所示。

(7) 选择“View”→“Logic Analyzer Window”命令，打开“Logic Analyzer”窗口。

(8) 在“Logic Analyzer”窗口，单击“Setup...”按钮，打开“Setup Logic Analyzer”对话框，如图 21-32 所示。在“Current Logic Analyzer Signals:”文本框中输入变量 DAC0V 和 DAC1V。

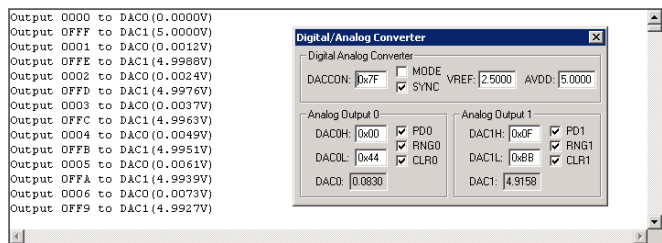


图 21-31 D/A 转换器仿真结果

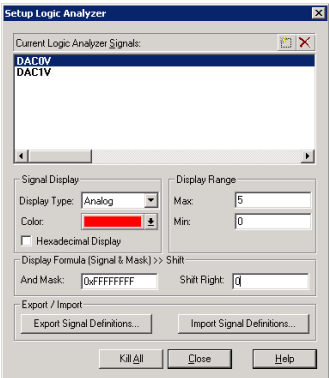


图 21-32 “Setup Logic Analyzer”对话框

(9) 单击“Close”按钮，关闭“Setup Logic Analyzer”对话框。此时，在“Logic Analyzer”窗口，便可以看到 DAC0V 和 DAC1V 变量，即通过 D/A 转换输出的模拟电压波形，如图 21-33 所示。

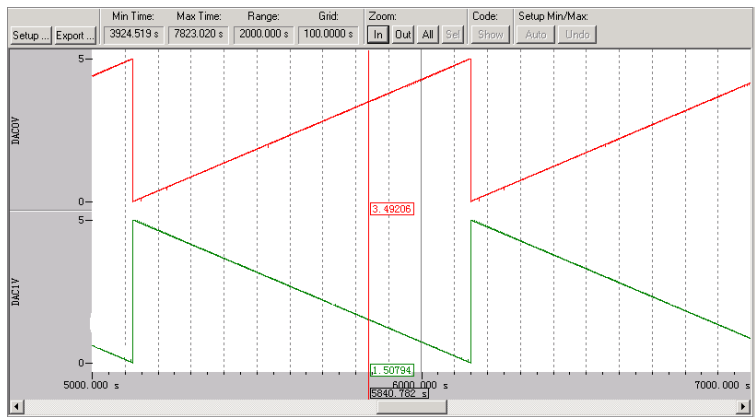


图 21-33 D/A 转换输出的模拟电压仿真图

(10) 当仿真操作完毕后，选择“Debug”→“Stop Running”命令，便可以结束程序的仿真运行。

(11) 选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。

在以上的仿真操作中，也可以选择“Debug”→“Step”命令或“Debug”→“Step Over”命令来单步执行程序，来进行程序仿真。

21.9 寄存器仿真

除了单片机各种片上资源外，Keil μ Vision3 集成开发环境还可以仿真单片机内部的各种寄存器。在程序仿真执行的时候，每一时刻寄存器中的值都可以查看，这样便于跟踪程序每一步执行的正确性。

21.9.1 寄存器的仿真界面

在 Keil μ Vision3 集成开发环境中，寄存器的仿真界面，如图 21-34 所示。
在寄存器的仿真界面上，实时显示了当前运行的内部寄存器的工作状态和数值。下面分别进行介绍其中各项的含义。

- “r0” ~ “r7”：通用工作寄存器 R0~R7。
- “a”：累加器 A，是运算过程中的暂存寄存器。
- “b”：寄存器 B，作为一般寄存器或中间结果暂存器使用。
- “sp”：堆栈指针寄存器，专门存放堆栈栈顶的地址。
- “sp_max”：堆栈指针最大值，用来观察是否产生堆栈溢出。
- “dptr”：数据指针寄存器，用来存放单片机片内 ROM 的地址，也可以存放片外 RAM 和片外 ROM 的地址。
- “PC \$”：程序计数器，用来存放下一条需要执行指令的内存地址。
- “states”：程序运行状态数。
- “sec”：已运行时间，单位是秒。
- “psw”：程序状态寄存器，用于存放指令执行后的有关状态。其中各状态标志位的含义如下：“p”是奇偶校验标志位，“f1”是保留位，“ov”是溢出标志位，“rs”是工作寄存器选择标志位，“f0”是用户标志位，“ac”是辅助进位标志位，“cy”是高位进位标志位。

Register	Value
Regs	
r0	0x00
r1	0x00
r2	0x00
r3	0x00
r4	0x00
r5	0x00
r6	0x00
r7	0x00
Sys	
a	0x00
b	0x00
sp	0x07
sp_max	0x07
dptr	0x0000
PC \$	C:0x000E
states	389
sec	0.00014145
psw	0x00
p	0
f1	0
ov	0
rs	0
f0	0
ac	0
cy	0

图 21-34 寄存器的仿真界面

在程序仿真执行的时候，可以在寄存器的仿真界面上，实时观察各个寄存器的工作状态和值，从而了解程序的运行情况。

21.9.2 寄存器的仿真操作

对于 C51 语言程序来说，由于没有直接对寄存器进行的操作，因此寄存器仿真界面的值并不能直接和程序运行情况对应；而对于汇编语言的程序，其直接对寄存器进行操作，此时，便可以单步执行查看每一步执行的情况。因此，Keil μ Vision3 集成开发环境对寄存器的仿真支持，大大方便了汇编程序的调试。

- 下面以通用工作寄存器的操作为例，介绍寄存器仿真的步骤。具体仿真操作步骤如下：
- (1) 按照前面的方法，新建一个工程。其中，选择 Atmel 公司的 AT89S52 单片机。
 - (2) 选择“File”→“New”命令，新建一个程序文件，并保存为*.ASM 文件，可以在其中输入如下的程序代码。

```
ORG      0000H
JMP      START
ORG      1000H

START:   MOV      R0,#00H      ;立即寻址方式赋值
        MOV      R1,#11H      ;立即寻址方式赋值
        MOV      R2,#22H      ;立即寻址方式赋值
        MOV      R3,#33H      ;立即寻址方式赋值
        MOV      P0,R0        ;寄存器 R0 寻址方式写端口 P0
        MOV      P1,R1        ;寄存器 R1 寻址方式写端口 P1
        MOV      P2,R2        ;寄存器 R2 寻址方式写端口 P2
        MOV      P3,R3        ;寄存器 R3 寻址方式写端口 P3
        JMP      START        ;跳转
END
```

在该程序中，首先使用立即寻址方式为寄存器 R0、R1、R2 和 R3 赋值。然后使用寄存器

寻址方式将寄存器 R0、R1、R2 和 R3 中的数值直接输出到单片机的 P0、P1、P2 和 P3 端口上。

(3) 选择 “Debug” → “Start/Stop Debug Session” 命令，进入程序仿真调试环境。此时在集成开发环境的左侧便可以看到寄存器的仿真界面。

(4) 选择 “Peripherals” → “I/O-Ports” → “Port 0” 命令，打开并行 I/O 端口 P0 的仿真界面。以同样的步骤，打开并行 I/O 端口 P1、P2 和 P3 的仿真界面。

(5) 按 F10 键，开始单步执行程序。当执行到为通用寄存器赋值的指令时，可以看到寄存器仿真界面上对应的寄存器值改变，如图 21-35 所示。

(6) 继续按 F10 键，当执行到为并行 I/O 端口赋值的指令时，在并行 I/O 端口仿真界面上可以看到对应的改变，如图 21-36 所示。

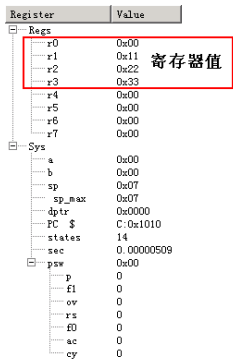


图 21-35 寄存器的仿真结果

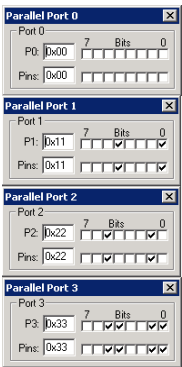


图 21-36 并行 I/O 端口的仿真结果

(7) 当仿真操作完毕后，可以选择 “Debug” → “Start/Stop Debug Session” 命令，退出程序仿真调试环境。

在前面的单步仿真执行过程中，还可以看到每执行一步，程序计数器 PC、状态 states 和程序运行时间 sec 均随之改变。寄存器仿真界面上显示的状态，和真实运行结果完全一致。

21.10 低功耗仿真

51 系列单片机提供了两种低功耗模式，省电保持模式和休眠运行模式。在程序设计中，可以通过特殊功能寄存器 PCON 来设置，说明如下：

- 当 CPU 置 PCON.1 位为 1 后，即 PD=1，系统便进入省电模式。
- 当 CPU 置 PCON.0 位为 1 后，即 IDL=1，系统便进入休眠运行模式。
- 若同时写入 PD 和 IDL 时，PD 优先，即优先进入省电模式。

另外，也可以在软件中通过检查 PCON.4 的标志位查看电源的状态，从而根据情况将单片机设置为低功耗模式。

在 Keil μ Vision3 集成开发环境中，提供了低功耗模式的仿真，但是没有提供相应的对话框。下面以 AT89S52 为例，介绍低功耗模式的程序设计以及仿真操作。具体仿真操作步骤如下：

(1) 按照前面的方法，新建一个工程。其中，选择 Atmel 公司的 AT89S52 单片机。

(2) 选择 “File” → “New” 命令，新建一个程序文件，并保存为 *.C 文件，可以在其中输入如下的程序代码。

```
#include<reg52.H> //头文件
#include<stdio.h>

void main(void) //主函数
{
while(1) //主循环
{
```

```
if (P1==0x01)                                //判断
{
    PCON=0x01;                               //省电模式
}
else if (P1==0x02)
{
    PCON=0x02;                               //休眠运行模式
}
Else                                         //其他语句
{
}
}
```

在该程序中，通过对 P1 端口的判断来使单片机进入不同的低功耗模式。当 P1 端口为 0x01 时，PCON=0x01，即进入省电模式；当 P1 端口为 0x02 时，PCON=0x02，即进入休眠运行模式。

- (3) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (4) 选择“Peripherals”→“I/O-Ports”→“Port 1”命令，打开并行 I/O 端口 P1 的仿真界面。
- (5) 在 P1 端口的仿真界面上设置 P1 端口的数据为 0x02，即模拟休眠按键输入。如图 21-37 所示。

(6) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时由于 P1 端口为 0x02，因此，程序中执行 PCON=0x02 语句，单片机进入休眠运行模式。在“Output Window”窗口可以看到“Power Down Mode invoked. Continue with RESET”的输出信息，表示单片机已经进入相应的低功耗模式，如图 21-38 所示。

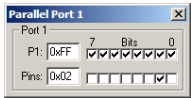


图 21-37 P1 端口置为 0x02

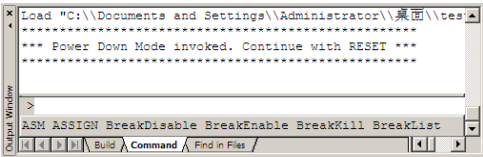


图 21-38 仿真输出结果

(7) 当仿真操作完毕后，可以选择“Debug”→“Start/Stop Debug Session”命令，退出程序仿真调试环境。

21.11 小结

本章详细介绍了 Keil μ Vision3 集成开发环境中，对单片机各种常见的片上资源的仿真操作。主要包括并行 I/O 端口、定时器/计数器、串行接口、中断、看门狗定时器、A/D、D/A、寄存器及低功耗仿真。单片机的程序设计主要是对各种片上资源进行操作，Keil μ Vision3 集成开发环境对各种片上资源均提供了完善的仿真支持。在程序设计时，通过仿真操作可以完美地模拟程序的执行情况，便于及时发现问题。这样便大大提高了程序开发的可靠性，加速了程序的开发速度。因此，读者应该熟练掌握本章内容。

第 22 章 Keil μ Vision3 中的 程序调试

在 Keil μ Vision3 集成开发环境中，除了能够仿真单片机各种片上资源和寄存器外，还具有完备的程序调试器。通过 Keil μ Vision3 的程序调试器，可以对程序进行单步调试、断点调试、代码覆盖分析及性能分析和优化等。本章将主要介绍 Keil μ Vision3 的程序调试器的各种调试功能，包括性能分析器、代码覆盖分析器和断点等。同时本章还将详细介绍 Keil μ Vision3 的各种调试指令。

22.1 Keil μ Vision3 的程序调试器概述

Keil μ Vision3 集成开发环境具有完备的程序调试功能，结合硬件资源的仿真，便可以实现无须外部硬件的完善仿真调试。Keil μ Vision3 的程序调试器除具有最基本的调试功能外，还包含一些高级调试工具才具有的代码覆盖分析等功能。

Keil μ Vision3 程序调试器的主要特性如下：

- 完整的符号信息。
- 源代码级别的调试。
- 断点调试功能。
- 带有条件的复杂断点。
- 扩充的 C 语言调试功能。
- 性能分析器。
- 代码覆盖分析器。
- 两个 Watchpoint 窗口。
- 完善的调试命令。
- 多样化的调试函数。
- 和外部硬件无缝接口，实现程序的在线调试。

基本的仿真调试在前面一章已有所介绍，下面将介绍其他一些常用的调试功能、调试命令及调试技巧。这些调试操作都可以在不使用外部硬件的情况下，对程序进行完备的分析，快速发现并解决问题。

22.2 性能分析器

Keil μ Vision3 集成开发环境中内建了性能分析器。其可以在程序运行时，统计各个函数或程序模块的执行次数及运行时间。这样，通过性能分析器的统计结果，便可以找到程序最耗时的部分，进行优化。使用性能分析器的步骤如下：

- (1) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (2) 选择“View”→“Performance Analyzer Window”命令，打开性能分析器窗口，如图 22-1 所示。
- (3) 在性能分析器窗口，右键单击选择“Setup PA”命令，打开“Setup Performance Analyzer”对话框，如图 22-2 所示。

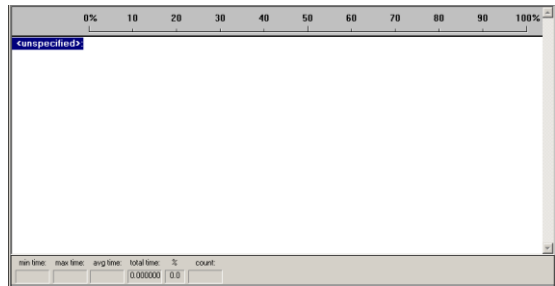


图 22-1 性能分析器窗口

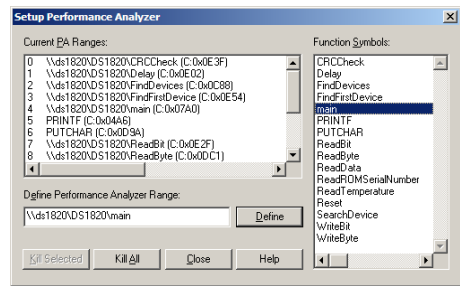


图 22-2 “Setup Performance Analyzer”对话框

- (4) 在“Setup Performance Analyzer”对话框的“Function Symbols”中双击选择需要分析的函数。
- (5) 单击“Define”按钮，将该函数添加到“Current PA Ranges”中。
- (6) 单击“Close”按钮，关闭“Setup Performance Analyzer”对话框。
- (7) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时，在性能分析器窗口便可以看到性能分析器的统计结果，如图 22-3 所示。

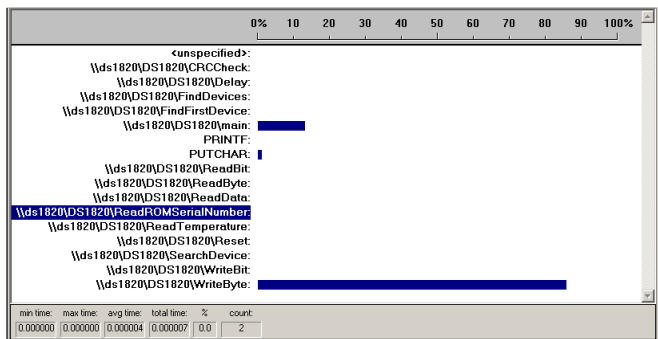


图 22-3 性能分析器统计结果

在性能分析器窗口中，显示了各个函数执行所消耗单片机 CPU 时间的比例。同时选择某个函数，还可以看到该函数所执行的次数等信息。

22.3 代码覆盖分析器

Keil μ Vision3 集成开发环境中内建了代码覆盖分析器。其可以在程序运行时，统计已执行的程序代码的比例，结果以百分数显示。使用性能分析器的步骤如下：

- (1) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (2) 选择“View”→“Code Coverage Window”命令，打开代码覆盖分析器窗口，如图 22-4 所示。其中列出了主要代码的执行情况。
- (3) 选择“Debug”→“Run”命令，程序便开始仿真执行。此时，在代码覆盖分析器窗口便可以看到代码执行统计结果，如图 22-5 所示。

通过代码覆盖分析器，可以很方便地看到哪些代码已经执行过，哪些代码没有执行。这样，根据这些信息便可以对程序的执行进行系统的分析。

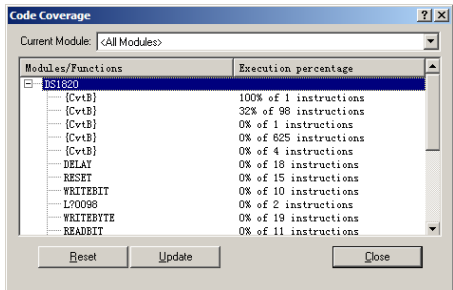


图 22-4 代码覆盖分析器窗口

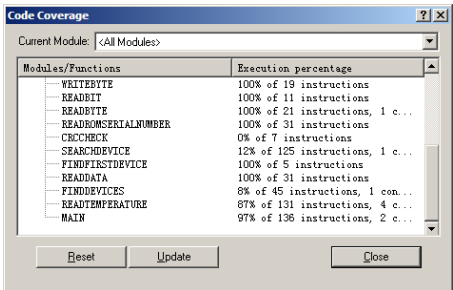


图 22-5 代码覆盖分析结果

22.4 断点

Keil μ Vision3 集成开发环境中内建了完善的断点调试功能。最简单的断点可以双击源代码的某一行，来放置一个断点。断点以代码行号前的有色方块（红色）显示，如图 22-6 所示。简单断点也可以通过工具栏、右键菜单或主菜单来实现。

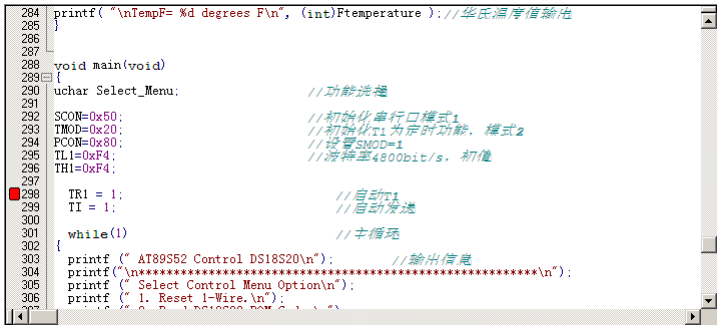


图 22-6 设置简单断点

对于一些复杂的程序调试，经常需要使用带有条件表达式及计数触发的断点。对于这些复杂断点的操作步骤如下：

- (1) 首先双击源代码的某一行，设置一个断点。
- (2) 选择“Debug”→“Start/Stop Debug Session”命令，进入程序仿真调试环境。
- (3) 选择“Debug”→“Breakpoints”命令，打开“Breakpoints”对话框，如图 22-7 所示。

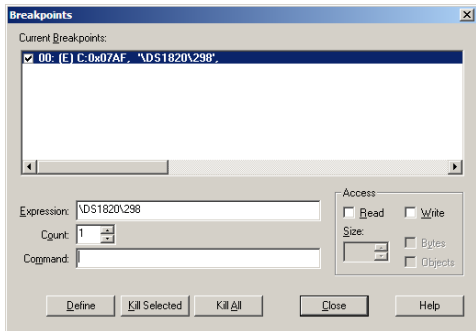


图 22-7 “Breakpoints”对话框

在“Breakpoints”对话框中，可以定义新断点。断点也可以设置成由计数触发，当计数减少到 0 时才触发断点。同样，在该对话框中也可以设置带有条件的断点表达式。

22.5 Keil μVision3 调试命令

Keil μ Vision3 支持完善的调试命令，这些调试命令可分为以下几大类。

- 通用命令：提供多种调试操作。例如进行系统复位，退出调试模式等操作。
- 程序命令：用于执行目标程序，分析程序运行性能。例如跳出当前函数，停止程序运行等操作。
- 断点命令：用于进行断点操作。断点命令用来控制断点，例如开启或关闭断点，删除或添加断点等。当程序运行到某条指令时，可以通过断点停止程序运行，此时可以执行调试命令或执行用户程序。
- 存储器命令：用来显示和更改存储器的内容。例如显示存储器中的内容，在线编译代码等操作。

在 Keil μ Vision3 中，调试命令只在程序运行或调试状态下使用。在命令执行时，可以输入命令的完整表示，也可以输入简写指令。例如，要输入 BreakSet 命令设置断点时，也可以输入其简写形式 BS，执行效果完全相同。下面分别介绍各类调试指令。

22.5.1 通用命令

通用命令提供多种调试操作，使用通用命令使得调试程序更加灵活。在 Keil μ Vision3 中提供的通用命令，如表 22-1 所示。

表 22-1 通用命令

命 令	描 述	命 令	描 述
ASSIGN	为串行窗口分配输入输出源	MODE	设置 PC 串行口的参数
DEFINE	创建一个工具箱按钮	RESET	复位操作
DIR	列出程序中的变量	SAVE	保存存储器内容
EXIT	退出调试模式	SCOPE	显示程序中的模块和函数的地址分配
INCLUDE	执行命令文件中的命令	SET	设置预定义变量的字符串值
KILL	删除调试函数和工具箱按钮	SIGNAL	显示信号函数状态和删除已启动的信号函数
LOAD	加载目标模块	SLOG	对串行窗口日志文件进行操作
LOG	调试窗日志文件操作		

各个通用命令的具体用法如下：

1. ASSIGN 命令

ASSIGN 命令用来显示和改变串口的输入输出。在命令窗口，输入 ASSIGN 命令并按 Enter 键执行后，将显示串口输入输出的配置，如图 22-8 所示。

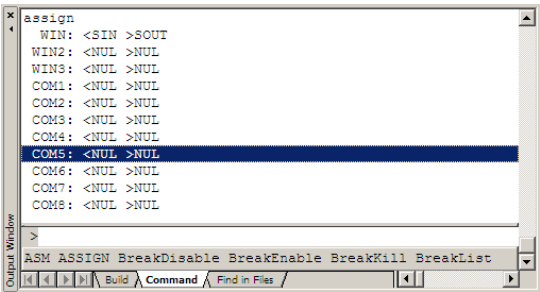


图 22-8 ASSIGN 命令

Keil μ Vision3 集成开发环境支持 WIN、WIN2、WIN3 和 COM1~COM8 串口。其中 WIN、WIN2 和 WIN3 为单片机的串行窗口，COM1~COM8 为 PC 机的串口。通过该命令可以仿真单片机和 PC 机的串口通信。

使用“ASSIGN channel<inreg>outreg”命令，可以改变串口输入输出的设置。

2. DEFINE 命令

DEFINE 命令用来创建一个带类型的变量，该变量可以被赋值。其命令格式如下：

```
DEFINE type identifier
```

该指令按指定的类型定义一个名为 identifier 的变量。其中，identifier 是变量名，它必须符合变量及标号的定义规则。使用 DEFINE 命令创建的变量可以用来装载编译器函数的返回值，也可以指定编译器函数的输入值。该变量可以像其他公用变量一样使用，但是其不占用仿真或目标 CPU 的存储器。

DEFINE 命令也可以用来创建工具箱按钮，其格式如下：

```
DEFINE BUTTON "label", "cmd"
```

该指令定义一个工具箱按钮。其中，label 是分配按钮的名称，cmd 是编译器命令或分配给按钮的命令。选择“View”→“Toolbox”命令，可以打开工具箱，在其中单击按钮就可执行该命令。

3. DIR 命令

DIR 命令用来显示各种类型的符号变量。该命令支持的操作有如下几种。

- DIR：显示当前模块的变量名称。程序计数器指向地址的程序属于哪个模块，该模块就是当前模块。
- DIR \module：显示当前指定模块的变量名称。
- DIR \module LINE 或 DIR \module\func LINE：显示指定模块中或模块中 func 函数的行号信息。
- DIR \module\func：显示指定模块中 func 函数的变量名称。
- DIR BFUNC：显示所有预定义的编译器函数的名称。
- DIR DEFSYM：显示用 DEFINE 命令创建的变量。
- DIR FUNC：显示 Keil μ Vision3 所有编译器函数的名称。
- DIR LINE：显示当前模块的行数。
- DIR MODULE：显示目标程序的所有模块名称。
- DIR PUBLIC：显示所有全局变量的名称。
- DIR SIGNAL：显示用户自定义的信号函数的名称。
- DIR UFUNC：显示用户自定义的函数的名称。

➤ DIR VTREG: 显示当前单片机所支持的引脚寄存器的名称。

以显示 Keil μ Vision3 编译器所有的函数为例。在命令窗口输入“DIR FUNC”命令，将罗列出所有的调试函数，如图 22-9 所示。

4. EXIT 命令

EXIT 命令用于停止 Keil μ Vision3 编译器的调试器模式。在命令窗口输入“EXIT”命令，并按 Enter 键，此时程序退出调试模式。该命令和“Debug”→“Start/Stop Debug Session”菜单命令一样。

5. INCLUDE 命令

INCLUDE 命令用来指定一个文件，并从文件中按行读出命令，让编译器执行。其命令格式如下：

```
INCLUDE path filename
```

其中，filename 是文件名。该命令读取文件或按编译器命令操作该文件。

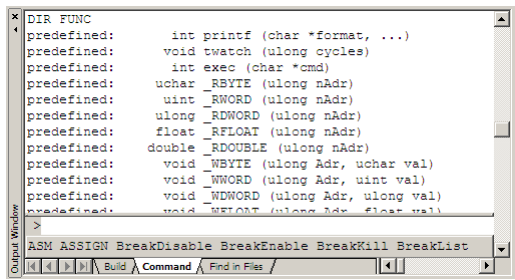


图 22-9 DIR FUNC 命令

6. KILL 命令

KILL 命令用来删除前面定义的工具箱按钮和编译器函数。该命令支持的操作有如下几种。

- KILL BUTTON number: 删除一个工具箱按钮。
- KILL FUNC*: 删除所有用户编译器函数。
- KILL FUNC function_name: 删除名为 function_name 的编译器函数。

其中，number 为工具箱按钮的序号，function_name 为用户编译器函数。这里，按钮的序号显示在工具箱窗口每个按钮的前面。使用该命令来删除用户编译器函数时，不会删除编译器预定义的函数。

7. LOAD 命令

LOAD 命令用来让编译器载入一个指定的文件。其命令格式如下：

```
LOAD path filename NOCODE
```

该命令载入一个绝对目标文件或 HEX 文件。其中，NOCODE 表示编译器仅载入符号信息，忽略代码记录。

LOAD 命令可以在编译器调试开始运行时，就载入当前项目的目标文件。

8. LOG 命令

LOG 命令用来创建、添加、检查状态和关闭一个日志文件。该命令支持的操作有如下几种。

- LOG > path filename: 创建一个名为 filename 的日志文件。命令窗口的输出信息将写入该文件。
- LOG >> path filename: 打开一个已经存在的名为 filename 的日志文件，并在后面添加

日志信息。如果该文件不存在，将创建文件。命令窗口的输出信息将添加到该文件中。

- LOG: 显示当前日志文件状态。
- LOG OFF: 关闭当前日志文件。

LOG 命令执行后，显示在命令窗口的输出信息会记录到日志文件中。下面举例说明该命令的用法，在命令窗口依次输入如下的命令。

```
LOG > C:\LogTest.txt
LOG
D
LOG OFF
```

这里首先创建一个日志文件 LogText.txt，然后使用 LOG 命令查询日志文件的状态。接着，使用 D 命令显示存储器中的内容，并输出到日志文件中。最后，使用 LOG OFF 命令关闭日志文件。这些命令的执行结果，如图 22-10 所示。



图 22-10 LOG 命令

9. MODE 命令

MODE 命令用来用来改变 PC 机的串口设置。其命令格式如下：

```
MODECOMx,baudrate,parity,databits,stopbits
```

其中，baudrate 参数是波特率值，如 1200，2400，9600 或 19200；parity 表示奇偶校验位，为 0 时表示无校验位，1 为奇校验，2 为偶校验；databits 表示有效数据位，为 8 时，表示有 8 个数据位，为 7 表示有 7 个数据位；stopbits 表示停止位，为 1 时表示 1 个停止位；为 1.5 时表示 1¹/₂ 个停止位；为 2 时表示 2 个停止位。

10. RESET 命令

RESET 命令用来复位调试系统，该命令支持的操作有如下几种。

- RESET: 复位仿真或目标。此时，该命令和工具栏的“Reset CPU”按钮功能一致。
 - RESET MAP: 复位寄存器映射 MAP 分配，以及清除任何载入的目标程序的存储器和调试信息。
 - RESET variable: 复位 SET 变量。
- 其中，variable 为使用 SET 来设置的变量。

11. SAVE 命令

SAVE 命令用来保存一个存储器映像到文件中。其命令格式如下：

```
SAVE path filename addr1,addr2
```

其中，path 为路径，filename 为文件名，addr1 为起始地址，addr2 为末尾地址。该文件以 HEX386 格式保存 addr1 至 addr2 的存储器内容，但是不能保存单片机内部 RAM 的 0x80 至 0xFF

之间的内容。保存在文件里的内容,可以通过 LOAD 命令再次载入到 Keil μ Vision3 的调试器中。

12. SCOPE 命令

SCOPE 命令用来显示指定模块或函数的地址范围。其命令格式如下:

```
SCOPE \module\function
```

该命令可以显示目标程序或一个程序模块的所有函数的地址范围,还可以显示单个函数的地址范围。

13. SET 命令

SET 命令用来设置与预定义变量相关联的字符串。其命令格式如下:

```
SET variable="string"
```

其中, variable 为指定的 SET 变量, 该命令将字符串 string 赋给 variable。

SET 命令也可以用来显示当前与一个与预定义变量相关联的字符串。其命令格式如下:

```
SET variable
```

其中, variable 为变量名, 该命令显示一个 SET variable 的路径分配情况。

14. SIGNAL 命令

SIGNAL 命令用来显示或停止已启动的信号函数, 该命令支持的操作有如下几种。

- SIGNAL KILL function_name: 停止指定的已启动的信号函数。
- SIGNAL STATE: 显示已启动的信号函数。

15. SLOG 命令

SLOG 命令用来创建、添加、检查状态和关闭一个日志文件。该命令支持的操作有如下几种。

- SLOG > path filename: 创建一个名为 filename 的日志文件。串口仿真窗口的输出输入信息将写入到该文件。
- SLOG>>path filename: 打开一个已经存在的名为 filename 的日志文件, 并添加日志信息。如果该文件不存在, 将创建文件。串口仿真窗口的输出输入信息将添加到该文件中。
- SLOG: 显示当前日志文件状态。
- SLOG OFF: 关闭当前日志文件。

SLOG 命令和 LOG 命令类似, 只不过 SLOG 命令用来保存串行仿真窗口的信息。

22.5.2 程序命令

程序命令可以在命令窗口运行代码和程序指令, 使用程序命令大大提高了程序调试的灵活性和方便性。程序命令如表 22-2 所示。

表 22-2 程序命令

命 令	描 述	命 令	描 述
COVERAGE	显示当前代码运行覆盖情况	PSTEP	跳跃执行指令, 不步入程序块或函数中
Esc	停止程序运行	OSTEP	跳出当前函数
Go	开始程序运行	TSTEP	跳跃执行指令, 步入函数中
Performance Analyzer	启动性能分析器		

各个程序命令的具体用法如下:

1. COVERAGE 命令

COVERAGE 命令用来显示程序代码的运行覆盖情况, 该命令支持的操作有如下几种。

- **COVERAGE**: 显示整个应用程序代码的运行覆盖情况。
 - **COVERAGE \module**: 显示所选择的代码模块运行覆盖情况。
 - **COVERAGE \module\func**: 显示所选择的 **func** 函数的运行覆盖情况。
- COVERAGE** 命令与 **LOG** 命令配合使用，可以将运行覆盖情况记录到日志文件中。

2. **Esc** 命令

Esc 命令用于使编译器停止运行目标程序。当 **Esc** 命令执行后，目标程序停止运行，同时寄存器窗口、变量监视窗口、反汇编窗口，以及其他仿真窗口会被更新，以反映最新的 CPU 运行状态。

3. **GO** 命令 (**G**)

GO 命令用来使程序开始运行。其命令格式如下：

```
GO startaddr,stopaddr
```

其中，**startaddr** 为起始地址，**stopaddr** 为末尾地址。当 **GO** 命令执行后，将从指定的地址 **startaddr** 开始运行程序，到地址 **stopaddr** 停止。对于指定 **stopaddr** 的位置，Keil μ Vision3 编译器将设置一个临时断点，程序运行到 **stopaddr** 后，程序停止并自动删除断点。

如果没有指定 **startaddr**，则从当前程序计数器开始运行，目标程序停止在 **stopaddr** 地址上。如果没有指定 **stopaddr**，程序只有运行到一个断点或手动单击停止按钮时，才会停止运行。

同 **Esc** 命令一样，使用 **GO** 命令后，目标程序运行停止在 **stopaddr** 处，此时寄存器窗口、变量监视窗口、反汇编窗口及其他仿真窗口会被更新，以反映最新的 CPU 运行状态。

4. **OSTEP** 命令 (**O**)

OSTEP 命令用于跳出当前函数。其命令格式如下：

```
OSTEP
```

OSTEP 命令从程序计数器的当前值开始运行，并停止在当前调用函数的指令的后一个指令上。如果没有调用函数，编译器会报告产生错误。

5. **Performance Analyzer** 命令 (**PA**)

Performance Analyzer 命令用于启动并管理性能分析器，该命令支持的操作有如下几种。

- **Performance Analyzer**: 显示所有当前定义的性能分析段的地址、索引号、执行次数、最小及最大和总执行时间（用时钟周期表示）。
- **Performance Analyzer start,end**: 定义一个新的性能分析段。其中，如果 **start** 为函数名，编译器会自动获取函数的开始和结束地址；如果 **start** 为地址，则 **end** 就必须是指定段的结束地址。
- **Performance Analyzer KILL***: 删除当前所有的性能分析段。
- **Performance Analyzer KILL* item,item,...**: 删除指定的性能分析段，其中 **item** 为性能分析段的索引号。该命令将从性能分析器中删除与索引号对应的地址段。
- **Performance Analyzer RESET**: 复位性能分析器，并删除所有已定义的地址段的记录信息。

所有这些命令也可以在主菜单选择“**Debug**”→“**Performance Analyzer**”命令，打开“**Setup Performance Analyzer**”对话框来设置。相关的操作可参考前面关于性能分析器的介绍。

使用性能分析器时，首先指定打算进行分析的部分，性能分析器在程序运行时收集这些部分的运行情况以进行统计。性能分析器最多可分析 256 段代码，记录每段代码的运行次数和运行时间。程序运行时，性能分析器的结果显示在性能分析器窗口中。使用性能分析器，可以帮助程序设计者调整程序达到最好的运行性能。

6. PSTEP 命令

PSTEP 命令跳跃执行指令，不步入程序块或函数中。该命令支持的操作有如下几种。

- PSTEP：跳过函数或子程序。
- PSTEP expression：执行或跳过 expression，expression 可以是一个程序行或一条汇编指令。

调试窗口有显示和汇编两种模式。依赖于调试窗口的显示模式，PSTEP 命令执行一条或多条源代码级指令或汇编指令。

7. TSTEP 命令

TSTEP 命令跳跃执行指令，步入函数中。该命令支持的操作有如下几种。

- TSTEP：步入函数或子程序。
- TSTEP expression：执行或单步运行 expression，expression 可以是一个程序行或一条汇编指令。

调试窗口有显示和汇编两种模式。依赖于调试窗口的显示模式，PSTEP 命令执行一条或多条源代码级指令或汇编指令。

22.5.3 断点命令

Keil μ Vision3 提供的断点命令可以管理调试断点。断点命令如表 22-3 所示。

表 22-3 断点命令

命 令	描 述	命 令	描 述
BREAKDISABLE	关闭断点	BREAKLIST	列举当前所有的断点
BREAKENABLE	开启断点	BREAKSET	设置一个断点表达式
BREAKKILL	从断点列表中删除断点		

除了执行断点命令来管理断点外，也可以通过菜单和对话框来设置断点，可以参阅前面断点的介绍。各个断点命令的具体用法如下：

1. BREAKDISABLE 命令（BD）

BREAKDISABLE 命令用来关闭一个或多个前面设定的断点。该命令支持的操作有如下几种。

- BREAKDISABLE number, number,...：关闭指定的断点，其中，number 是断点的序号。
- BREAKDISABLE*：关闭所有断点。

BREAKDISABLE 命令仅是关闭断点，并不删除断点，程序执行时将忽略该断点。

假设程序设置有两个断点。这里首先使用 BL 命令列举出所有的断点，然后输入如下命令：

BD 1

指令的执行情况，如图 22-11 所示。BD 指令执行后，将关闭断点 1。此时，断点 1 处将变成白色块来显示，表示该断点被关闭，程序将忽略该断点。如图 22-12 所示。

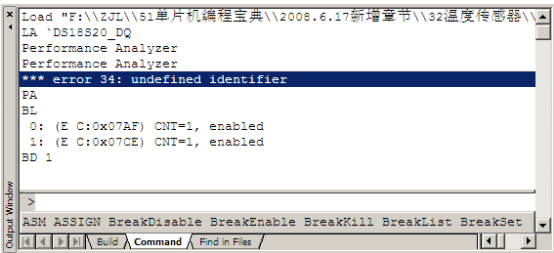


图 22-11 断点命令的执行

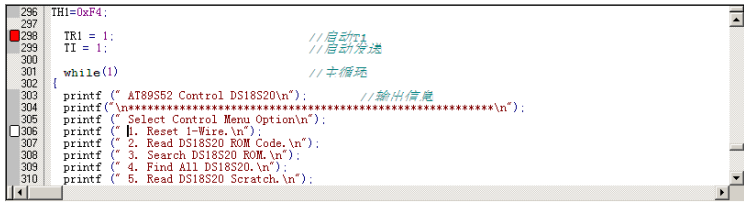


图 22-12 指定断点被关闭

2. BREAKENABLE 命令（BE）

BREAKENABLE 命令用来开启一个或多个断点。该命令支持的操作有如下几种。

- BREAKENABLE number, number,...: 开启指定的断点，其中，number 是断点的序号。
- BREAKENABLE*: 开启所有断点。

BREAKENABLE 命令主要用来开启由 BREAKDISABLE 命令关闭的断点。

3. BREAKKILL 命令（BK）

BREAKKILL 命令用来从断点列表中删除断点。该命令支持的操作有如下几种。

- BREAKKILL number, number,...: 删除指定的断点，其中，number 是断点的序号。
- BREAKKILL *: 删除所有断点。

BREAKKILL 命令用来删除断点，不同于 BREAKDISABLE 命令。

4. BREAKLIST 命令（BL）

BREAKLIST 命令用来列举出所有断点。其命令格式如下：

BREAKLIST

BREAKLIST 命令执行后，调试窗口将显示所有的断点。断点信息按以下格式显示。

number: (type)'expression',CNT=count,enable_flag
exc("command")

其中，number 表示断点索引号；type 表示断点类型；expression 表示断点定义的原始文字；count 表示断点通过次数，编译器在第 count 次运行到该断点时，停止程序运行或执行指定的指令。enable_flag 显示为 enabled 时表示是一个开启的断点，显示为 disabled 时为一个关闭的断点；command 表示当程序执行到断点时执行的命令。

这里，断点类型有执行断点（e：紧跟着地址）、条件断点（c）、存取断点（a：紧跟着“rd”表示读，“wr”表示写，“rw”表示读写，后跟地址）。

典型的 BREAKLIST 命令的执行结果，如图 22-13 所示。

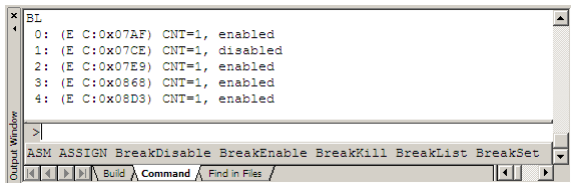


图 22-13 列举所有断点

5. BREAKSET 命令（BS）

BREAKSET 命令用来设置一个断点表达式。该命令支持的操作有如下几种。

- BREAKSET exp, cnt,"cmd": 设置一个执行或条件断点。其中，exp 为指定地址，或由编译器运行时计算的表达式；cnt 为断点通过次数，默认为 1；cmd 为命令字符串。
- BREAKSET READ exp, cnt,"cmd": 设置一个读断点。其中，exp 为指定读断点的地址，

或由编译器运行时计算的表达式；cnt 为断点通过次数，默认为 1；cmd 为命令字符串。

- **BREAKSET WRITE exp, cnt, "cmd"**：设置一个写断点。其中，exp 为指定写断点的地址，或由编译器运行时计算的表达式；cnt 为断点通过次数，默认为 1；cmd 为命令字符串。
- **BREAKSET READWRITE exp, cnt, "cmd"**：设置一个读写断点。其中，exp 为指定读写断点的地址，或者由编译器运行时计算的表达式；cnt 为断点通过次数，默认为 1；cmd 为命令字符串。

BREAKSET 命令用来在指定的指令设置一个断点，并通过 cmd 指定编译器在发生断点时执行的命令。如果不指定断点命令 cmd，在发生断点时，编译器仅停止程序的运行。当指定断点命令 cmd 时，编译器会执行 cmd 指令，但是目标程序不停止运行。

22.5.4 存储器命令

存储器命令可以用来显示或修改存储器内容。存储器命令如表 22-4 所示。

表 22-4 存储器命令

命 令	描 述	命 令	描 述
ASM	在线编译代码	MAP	存储器映射
DEFINE	定义在 Keil μ Vision3 函数中可用的类型变量	UNASSEMBLE	反汇编程序存储器
DISPLAY	显示存储器中的数据内容	WATCHSET	设置一个监视点到监视窗口
ENTER	输入数值到一个指定的存储器地址	WATCHKILL	清除所有的监视点
EVALUATE	计算表达式并输入结果		

各个存储器命令的具体用法如下：

1. ASM 命令

ASM 命令用来显示和设置当前汇编的地址，让用户输入汇编指令。该命令支持的操作有如下几种。

- **ASM**：显示当前汇编代码的地址。
- **ASM start address**：设置汇编代码的地址为 start address。
- **ASM instruction**：汇编指定的指令代码，并将结果码存入存储器中的当前汇编地址。当前汇编代码地址增加该指令的字节数。

ASM 命令可用来对正调试的目标程序做一些临时的修改。

2. DISPLAY 命令（D）

DISPLAY 命令用来在已打开的储存器窗口或命令窗口显示一段地址范围的储存器的内容。其命令格式如下：

```
DISPLAY startaddr,endaddr
```

其中，startaddr 为起始地址，endaddr 为末尾地址。该命令执行后，将显示从地址 startaddr 到地址 endaddr 范围储存器的内容。储存器区域显示为 HEX 或 ASCII 格式。

如果地址参数 startaddr 和 endaddr 被省略，储存器内容显示从上一个 **DISPLAY** 命令显示的存储器地址结尾处开始。如果是第一次使用 **DISPLAY** 命令，将从程序存储器的 0x0000 地址开始显示。

对于 8051 和 2051 类型的单片机，存储器地址必须加前缀以指定存储器类型。Keil μ Vision3 中所支持的存储器类型前缀，如表 22-5 所示。

表 22-5 存储器类型的前缀

存储器类型	说 明	存储器类型	说 明
B	位寻址 RAM 存储器 (BIT)	ED	扩展内部直接寻址 RAM 存储器 (EDATA)
C	代码存储器 (CODE)	HC	海量常量存储器 (HCONST)
CO	常量存储器 (CONST)	I	8051 内部间接寻址 RAM 存储器 (IDATA)
D	8051 内部直接寻址 RAM 存储器 (DATA)	X	外部数据存储器 (XDATA)
EB	扩展位寻址 RAM 存储器 (EBIT)		

例如，在命令窗口输入如下命令。

```
D c:0,0x60
```

该命令用于显示代码存储器从 0 到 0x60 之间的数据内容。如图 22-14 所示。

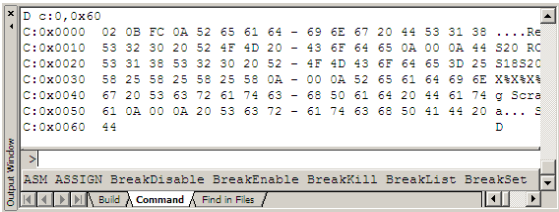


图 22-14 DISPLAY 命令

3. ENTER 命令 (E)

ENTER 命令用来输入数值到一个指定的存储器地址。其命令格式如下：

```
ENTER type address=expr,expr,...
```

其中，type 为数据类型；expr 为表达式。

ENTER 命令用来改变从指定地址开始的存储器的内容。可以指定多重表达式，由逗号“,”隔开。表达式将被转换为指定的数据类型，并保存在相应的存储器中。

4. MAP 命令

MAP 命令用来操作存储器映射。该命令支持的操作有如下几种。

- MAP: 显示当前的存储器映射。
- MAP start,end [READ WRITE EXEC][VNM]: 按指定的存取方式映射存储器段。其中，start 为起始段，end 为结束段。
- MAP start,end CLEAR: 清除一个 start 和 end 之间的存储器段映射。

Keil μ Vision3 编译器使用目标程序的变量信息，自动建立存储器映射，目标程序从而可以存取使用存储器。

这里，MAP 命令可以指定一个地址段，并指明存取方式，包括读 (READ)、写 (WRITE) 和执行 (EXEC)。存储器映射支持一字节长度的段。VNM 选项使得编译器将指定的存储段认定为冯·诺依曼式存储器结构。当使用 VNM 选项时，编译器会将外部数据存储器 and 代码存储器重叠，即对外部数据存储器的写操作也会改变代码存储器的内容。用 VNM 选项指定的地址范围可以不在代码区范围之内，可以小于 64KB，且指定的地址范围必须是外部数据区域。

如果使用不带参数的 MAP 命令时，将会显示目标程序当前的存储器映射。如图 22-15 所示。

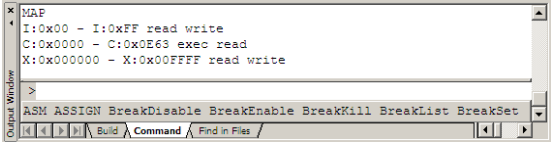


图 22-15 显示当前存储器映射

另外，RESET MAP 命令可以清除所有映射段，恢复到默认设置。

5. UNASSEMBLE 命令 (U)

UNASSEMBLE 命令用来对代码存储器进行反汇编，其命令格式如下：

```
UNASSEMBLE address
```

其中，address 表示从该位置开始反汇编。

UNASSEMBLE 命令执行后，将在 Disassembly 窗口显示反汇编代码。如果省略 address，则将从上次 UNASSEMBLE 命令的显示开始。

6. WATCHSET 命令 (WS)

WATCHSET 命令用来设置一个监视变量到监视窗口，其命令格式如下：

```
WATCHSET window #r, expression, base
```

其中，window #r 参数指定监视窗口的页号，默认状态下是变量观察窗口 1；expression 为观察点表达式；base 为指定使用的数字格式（十进制或十六进制），默认状态下是十六进制；

WATCHSET 命令定义显示在监视窗口的监视点的表达式。在每个程序命令执行后。监视窗口都将会更新显示监视点。

7. WATCHKILL 命令 (WK)

WATCHKILL 命令用于清除所有的监视点，该命令支持的操作有如下几种。

- WATCHKILL：删除所有监视点表达式。
- WATCHKILL window #r：删除变量观察窗口的一页中的所有观察点。其中 window #r 为指定的变量观察窗口页号，表示第 r 页。

22.6 小结

本章详细介绍了 Keil μ Vision3 集成开发环境的程序调试功能，包括性能分析器、代码覆盖分析器和断点等。本章还详细介绍了 Keil μ Vision3 编译器提供的调试命令，这些调试命令大大扩展了单片机程序的仿真调试。合理使用 Keil μ Vision3 的各种调试功能及调试命令，可以在程序设计和调试时达到事半功倍的效果。因此，熟练掌握本章内容有助于读者分析和优化单片机程序。

第五篇 典型案例篇

第 23 章 键盘程序设计

一个好的单片机应用系统，通常要有优秀的人机交互接口。键盘是与单片机进行人机交互的最基本的途径，其以按键的形式来设置控制功能或输入数据。按键的输入状态本质上是一个开关量。人们通过键盘输入一些命令或数据，以达到控制单片机运行的目的。

常用的键盘有独立式按键键盘和矩阵式按键键盘两种。独立式按键键盘接口简单，适合于简单而且少的开关量的输入。矩阵式按键键盘则适合于输入参数较多、功能复杂的系统，可以最大限度地使用单片机的引脚资源。本章将主要介绍独立式按键键盘和矩阵式按键键盘输入的相关知识及其编程接口。

23.1 键盘接口概述

键盘就是在人机交互系统中用来输入控制信号或数据的接口。其中，人机交互系统是一个完整的单片机系统的组成部分，用来使单片机识别不同的输入信号，并做出不同的响应。

对于一个优秀的人机键盘接口设计，需要占用合理的单片机资源，并能够及时、准确地响应用户的输入信息。在进行单片机键盘接口设计的时候，需要注意如下几个方面。

23.1.1 按键编码

按键的编码也就是每个按键在单片机程序设计时对应的键值。每个按键对应一个唯一的键值。当按键按下的时候，键盘将向单片机发送该按键对应的键值，单片机程序对不同的键值做出不同的响应。

在硬件上，键盘按键使用单片机的 I/O 线与 CPU 进行通信。其中单片机 I/O 线接收的是高低逻辑电平信号，因此，键盘输入的不同键值可以表示为 I/O 线上不同的高低电平的组合。键盘编码设计的主要任务就是选择合理的键盘结构，为每个按键分配不同的 I/O 输入信号，以供单片机识别并响应。

23.1.2 输入的可靠性

输入的可靠性即让单片机程序能够正确无误地响应按键操作。由于目前的键盘按键均为机械式接触点，由于触点的机械弹性效应，在按键闭合和断开的时候，接触点会出现抖动，这样可能导致误响应或多次响应等。键盘的可靠输入是键盘接口设计的关键点。对于键盘的可靠输入需要在程序中做如下两方面的处理。

- 去抖动。由于机械特性的不同，按键的抖动时间长短不等，大致在 5ms~10ms 之间。这样可以在硬件或软件中进行相应的处理来消除抖动的影响。
- 一次按键处理。由于人操作的按键闭合是有一定的时间限制的，一般来说，大致为 0.1s~5s。当按键按下之后，相应的按键编码以高低电平的方式输入到单片机的 I/O 口。因为单片机的执行速度很快，有可能导致单片机程序对该按键操作响应多次。

通常来说，采用延时程序可以同时达到去抖动和一次按键处理。采用这种方法时，当程序检

测到有键按下，便执行一个 10ms 的延时程序，然后再检测一次，看该键是否仍然闭合。如果仍然闭合则可以确认该按键确实被按下，从而可以消除抖动的影响，便可以执行相对应的操作。

采用延时程序的方法简单实用、而且成本低。当然也可以采用硬件防抖，一般来说，硬件处理比较复杂、成本也高，因此不推荐使用。

23.1.3 程序检测及响应

单片机对键盘输入的检测可以采用查询和中断两种方式。查询方式需要在程序中反复查询每一个按键的状态，因此会占用大量的 CPU 处理时间，这种方法适用于一般用途的程序。中断法是当有按键按下的时候向 CPU 申请中断，平时不会占用 CPU 处理时间，适用于一些对实时性要求较高的复杂单片机系统。

在程序中，对键盘的处理应该包括如下几个方面。

- 检测按键是否按下；
- 如果检测到按键被按下，执行延时程序，用来实现软件去抖动，消除抖动的影响；
- 扫描按键，准确判断按键的键值；
- 转向相应的程序处理子程序。

为了满足系统实时性的要求，程序对键盘输入的响应应该准确迅速。在按键对应的处理子程序中，不能执行过于繁重的任务而延误了对下一次按键动作的响应。

23.2 独立式按键及其编程接口

键盘有很多种类型，对于简单的系统，如果按键个数比较少、单片机资源比较宽裕，则可以使用独立式按键结构，这样可以简化程序设计。

23.2.1 独立式按键结构

独立式按键采用每个按键单独占有一个 I/O 口的结构，这是最简单的键盘输入设计。当按下和释放按键时，输入到 I/O 端口的电平是不一样的，单片机程序根据不同端口电平的变化判断是否有按键按下及是哪一个按键被按下，并执行相应的程序段。

51 单片机外接独立式按键的电路结构，如图 23-1 所示。其中按键和单片机引脚直接使用上拉电阻，当没有按键按下的时候，I/O 端口输入的是高电平，当按键按下的时候，I/O 端口输入的是低电平，从而实现端口电平的变化来达到按键输入的目的。

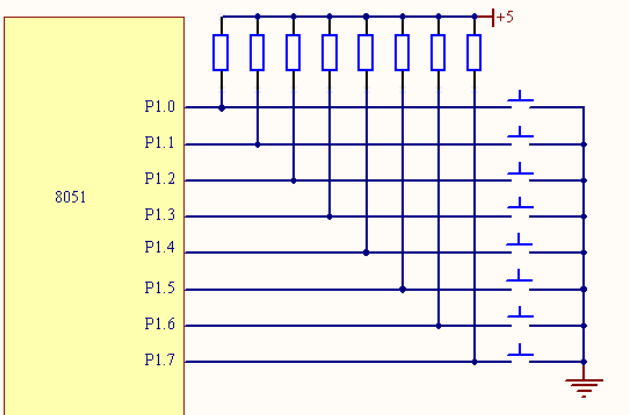


图 23-1 独立式按键的电路结构

这种独立式按键电路简单，方便程序处理。但是，由于每个按键都要单独占用一个单片机 I/O 引脚，因此不适用于按键输入较多的场合，这样会占用很多单片机 I/O 端口资源。

23.2.2 独立式按键程序设计

独立式按键的程序设计比较简单，一般采用查询方式即可。在程序设计时，可以采用汇编语言，也可以采用 C51 语言，下面分别给出采用这两种语言的程序范例。

如果采用汇编语言进行程序设计，则一般使用条件判断跳转指令（例如 JB 等），根据按键的状态使程序转向相应的代码段。程序示例如下：

```
START:  MOV      #0FFH      ;设置单片机的 P1 口为输入端口
        MOV      A, P1      ;读入 P1 端口的状态
        CPL      A
        JZ       START      ;若无按键按下，则返回重新读取扫描
        JB       ACC.0, F0    ;按键 0 按下时转向 F0
        JB       ACC.1, F1    ;按键 1 按下时转向 F1
        JB       ACC.2, F2    ;按键 2 按下时转向 F2
        JB       ACC.3, F3    ;按键 3 按下时转向 F3
        JB       ACC.4, F4    ;按键 4 按下时转向 F4
        JB       ACC.5, F5    ;按键 5 按下时转向 F5
        JB       ACC.6, F6    ;按键 6 按下时转向 F6
        SJMP     FUN7        ;按键 7 按下时转向 F7
F0:      AJMP     FUN0        ;跳转到子程序段 FUN0
F1:      AJMP     FUN1        ;跳转到子程序段 FUN1
F2:      AJMP     FUN2        ;跳转到子程序段 FUN2
F3:      AJMP     FUN3        ;跳转到子程序段 FUN3
F4:      AJMP     FUN4        ;跳转到子程序段 FUN4
F5:      AJMP     FUN5        ;跳转到子程序段 FUN5
F6:      AJMP     FUN6        ;跳转到子程序段 FUN6
F7:      AJMP     FUN7        ;跳转到子程序段 FUN7
FUN0:    .....              ;子程序段 FUN0
        LJMP     START      ;返回
FUN1:    .....              ;子程序段 FUN1
        LJMP     START      ;返回
FUN2:    .....              ;子程序段 FUN2
        LJMP     START      ;返回
FUN3:    .....              ;子程序段 FUN3
        LJMP     START      ;返回
FUN4:    .....              ;子程序段 FUN4
        LJMP     START      ;返回
FUN5:    .....              ;子程序段 FUN5
        LJMP     START      ;返回
FUN6:    .....              ;子程序段 FUN6
        LJMP     START      ;返回
FUN7:    .....              ;子程序段 FUN7
        LJMP     START      ;返回
```

如果采用 C51 语言进行程序设计，则可以使用循环语句来扫描按键。在下面的示例代码中，使用 while 语句来构成无限扫描循环。在循环中通过 switch 语句来根据按键来执行特定的代码。

程序示例如下：

```
while(1)
{
    switch(P1)
    {
        case 0xFE:           ;按键 0 按下 ( 11111110 )
            .....;
        case 0xFD:           ;按键 1 按下 ( 11111101 )
            .....;
        case 0xFB:           ;按键 2 按下 ( 11111011 )
            .....;
        case 0xF7:           ;按键 3 按下 ( 11110111 )
            .....;
        case 0xEF:           ;按键 4 按下 ( 11101111 )
            .....;
        case 0xDF:           ;按键 5 按下 ( 11011111 )
            .....;
        case 0xBF:           ;按键 6 按下 ( 10111111 )
            .....;
        case 0x7F:           ;按键 7 按下 ( 01111111 )
            .....;
    }
}
```

这里需要注意的是，根据查询顺序的不同，各个按键之间优先级的顺序不同，即当两个按键同时按下时，优先执行哪一个的问题。

23.3 4×4 矩阵式键盘及其编程接口

对于比较复杂的系统或按键比较多的场合，可用采用矩阵式键盘。矩阵式键盘有很多种，下面以应用最为广泛的 4×4 矩阵式键盘为例来介绍，其他矩阵式键盘的设计方法类似。

4×4 矩阵式键盘的结构，如图 23-2 所示。其由 4 根行线和 4 根列线交叉构成，按键位于行列的交叉点上，这样便构成 16 个按键。交叉点的行列线是不连接的，当按键按下时，此交叉点处的行线和列线导通。

在电路结构上，一般将行（X0~X3）和列（Y0~Y3）分别接到单片机的一个 8 位的并行端口上，程序中分别对行线和列线进行不同的操作便可以确定按键的状态。这样，只占用一个 8 位的并口便可以实现 16 个按键，因此矩阵式键盘对端口的利用率很高。

矩阵式键盘的工作方式有扫描法、线反转法和中断法三种。下面分别介绍各种方法是如何工作的。

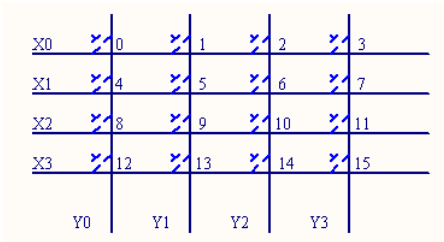


图 23-2 4×4 矩阵式键盘的结构

23.3.1 扫描法及其程序设计

扫描法是在程序中反复扫描查询键盘接口，根据端口的输入情况，调用不同的按键处理子程序。由于在执行按键处理子程序的时候，单片机不能再次响应按键请求。因此，单片机的按键处理子程序应该尽可能少占用 CPU 的运行时间，并且尽可能将键盘扫描安排在程序空余的时候，以满足实时准确响应按键请求的目的。

1. 扫描法的原理

在使用扫描法时，应将矩阵式键盘的行线通过上拉电阻接+5V 电源，如图 23-3 所示。此时

如果无任何按键按下，则对应的行线输出为高电平；如果有按键按下，对应交叉点的行线和列线断接，行线的输出依赖于与此行连接的列的电平状态。由此可以实现矩阵式键盘的编码处理。

键盘扫描法的流程图如图 23-4 所示。

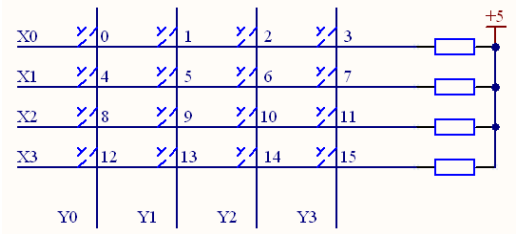


图 23-3 扫描法的电路结构

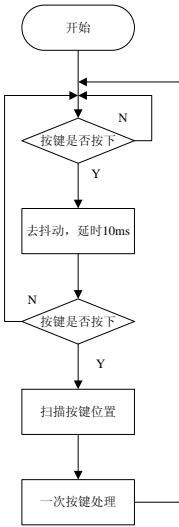


图 23-4 键盘扫描法的流程图

- 键盘扫描的一般步骤如下：
- (1) 判断键盘上有没有按键按下。将列线（Y0~Y3）全部输出为 0，此时读行线（X0~X3）的状态，如果行线全为 1，则表示此时没有任何按键按下；如果行线不全为 1，则表示有按键按下。进而继续执行下面的步骤。
 - (2) 按键软件去抖动。当判断有按键按下之后，程序中延时 10ms 左右的时间后，再次判断一下键盘的状态。如果仍然处于按键按下的状态，即行线不全为 1，则可以肯定有按键按下，否则当做按键的抖动来处理。
 - (3) 扫描按键的位置。先令列线 Y0 为低电平 0，其余三根列线均为高电平 1，此时读取行线的状态。如果行线（X0~X3）均为高电平，则 Y0 这一列上没有按键按下，如果行线（X0~X3）不全为高电平，则其中为低电平的行线与 Y0 相交的按键被按下。如果列 Y0 没有按键按下，这可以按照同样的方法依次检查列 Y1、Y2 和 Y3 有没有按键按下。这样逐列逐行扫描便可以找到按键按下的位置（X，Y）。
 - (4) 一次按键处理。有的时候，为了保证一次按键只进行一次按键处理，可以判断按键是否释放，如果按键释放则开始执行按键操作。

2. 扫描法程序设计

这里假定矩阵式键盘的 Y0~Y3 接单片机 P1.0~P1.3，而 X0~X3 接 P1.4~P1.7。由于矩阵式键盘比较复杂，因此一般采用 C51 语言来编写，这样可以使程序结构清晰。下面给出采用 C51 语言的键盘扫描子程序，示例如下：

```
int KeyScan()
{
    unsigned char k,k_temp;
    P1 = 0xf0;                                //低位置 0，准备查询按键
    k = P1;                                    //取得当前 P1 口的状态
    if(k != 0xf0)                             //如果有变化则表示有按键按下
    {
        delay();                             //延时，进行去抖动
        k_temp=P1;
```

```

if (k==k_temp)                                //确实有按键按下
{
    k = 0xfe;
    do                                         //循环扫描每一行
    {
        P1 = k;
        if(k != P1)
        {
            switch(P1)                        //判断按键，并返回键值
            {
                //第 1 行
                case 0xe: {return 0;break;}
                case 0xbe: {return 1;break;}
                case 0xde: {return 2;break;}
                case 0xee: {return 3;break;}
                //第 2 行
                case 0x7d: {return 4;break;}
                case 0xbd: {return 5;break;}
                case 0xdd: {return 6;break;}
                case 0xed: {return 7;break;}
                //第 3 行
                case 0x7b: {return 8;break;}
                case 0xbb: {return 9;break;}
                case 0xdb: {return 10;break;}
                case 0xeb: {return 11;break;}
                //第 4 行
                case 0x77: {return 12;break;}
                case 0xb7: {return 13;break;}
                case 0xd7: {return 14;break;}
                case 0xe7: {return 15;break;}
            }
        }
        k = _crol_(k,1);                      //移位，进入下一行扫描
    }while(k !=0xef);                          //超过列范围，退出扫描
}
}
}

```

23.3.2 线反转法及其程序设计

线反转法从本质上来说也是一种扫描法。在实际使用过程中，扫描法需要逐列扫描查询，根据键的位置不同，每次查询的次数不一样。如果按下的键位于最后一列时，则要经过多次扫描查询才能获得该按键的位置。而采用线反转法，无论被按得按键处于第一列还是最后一列，都只需要经过两步便可以获得此按键的位置。因此，线反转法更加方便。

1. 线反转法的原理

线反转法的原理图，如图 23-5 所示。线反转法的流程图，如图 23-6 所示。

利用线反转法的具体操作步骤如下：

(1) 将行线作为输出线，列线作为输入线。置输出线全部为 0，此时列线中呈低电平 0 的为按键所在的列，如果全部都不是 0，则没有按键按下。

(2) 将第一步反过来，即将列线作为输出线，行线作为输入线。置输出线全部为 0，此时行线中呈低电平 0 的为按键所在的列。至此，便确定了按键的位置 (X, Y)。

(3) 一次按键处理。有的时候为了保证一次按键只进行一次按键处理，可以判断按键是否释放，如果按键释放则开始执行按键操作。

同样，在实际应用中也应该采用软件延时的方法来进行去抖动处理。此时可以在第一步和

第二步之间加上延时语句，在第二步中即判断了是否抖动，也可以直接得到按键的位置。

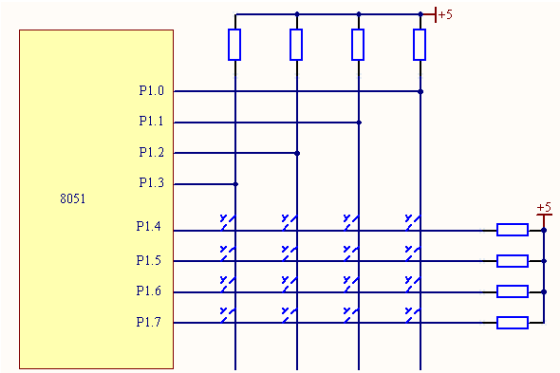


图 23-5 线反转法原理图

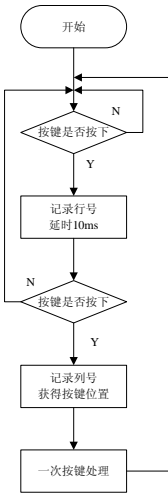


图 23-6 线反转法的流程图

2. 线反转法程序设计

下面采用 C51 语言实现线反转法的键盘扫描子程序，示例如下：

```
int KeyScan(void)
{
    int key;
    P1=0x0F;                                     //低 4 位输入
    delay();                                     //延时，用于去抖动
    temp=P1;                                     //读 P1 端口
    temp=temp&0x0F;                             //获取列值
    temp=~(temp|0xF0);
    if (temp==1)                                //根据列值，为 key 赋值
        key=0;
    else if (temp==2)
        key=1;
    else if (temp==4)
        key=2;
    else if (temp==8)
        key=3;
    else
        key=16;

    P1=0xF0;                                     //高 4 位输入
    delay();                                     //延时，用于去抖动
    temp=P1;                                     //读 P1 端口
    temp=temp&0xF0;                             //获取行号
    temp=~((temp>>4)|0xF0);
    if (temp==1)                                //根据行号，为 key 赋值
        key=key+0;
    else if (temp==2)
        key=key+4;
    else if (temp==4)
        key=key+8;
    else if (temp==8)
        key=key+12;
    else
```

```
key=16;

return key;           //返回按键编号
}
```

23.3.3 中断法及其程序设计

中断法是将键盘扫描程序放置在单片机的中断服务例程中的方法。对于扫描法和线反转法，其都是利用扫描查询的方式来获得按键信息，这样 CPU 总是要不断地扫描键盘，占用很多 CPU 处理时间。而中断法则只有当按键按下的时候，才触发中断，进而扫描键值。因此，采用中断法进行键盘设计可以提高 CPU 的工作效率，特别适合于复杂的系统或对实时性要求比较高的场合。

1. 中断法的原理

中断法的原理图，如图 23-7 所示。其中，4×4 矩阵式键盘的列线与单片机 P1 口的高 4 位相连，行线通过二极管与单片机 P1 口的低 4 位相连。P1.0~P1.3 作为输入端，P1.4~P1.7 作为输出端。键盘的 4 根行线分别引出连接到一个具有 4 输入端的与门，输出端接单片机的外部中断#INT0。

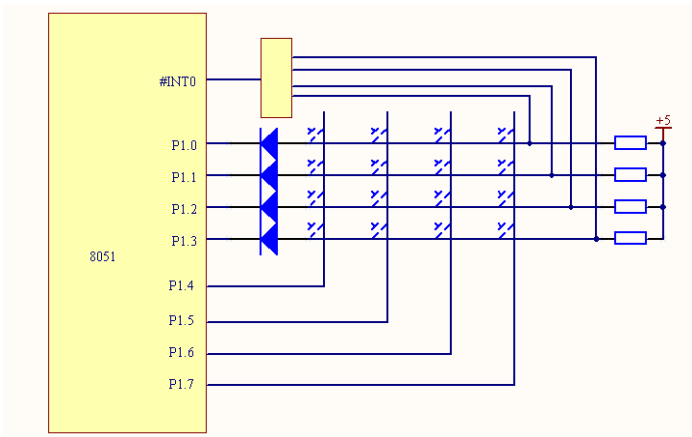


图 23-7 中断法原理图

系统初始化的时候，将键盘的输出端口全部置低电平 0。当有按键按下的时候，#INT0 将变为低电平，此时向 CPU 发出中断请求，CPU 响应中断并进入中断服务程序。在中断服务程序中，可以按照前面的扫描查询的方法来获得按键的位置信息 (X, Y)。

2. 中断法程序设计

按照中断法的原理，其只有在键盘上有键按下的时候，才发出中断请求。CPU 响应中断请求后，在中断服务程序中，进行键盘扫描，获得按键信息。下面给出 C51 语言编写的中断法键盘扫描的程序，示例如下：

```
#include <reg51.h>           //头文件
#include <intrins.h>

int KeyScan(void) ;          //键盘扫描函数
int key;

void main()
{
    //构造外部中断 0，INT0 ( P3.2 )
    IT0=1;                    //下降沿触发
```

```
EX0=1; //开 EX0 中断
EA=1; //开发中断
P1=0x0FF;
while(1) //用户程序
{
}

void FuncIr(void) interrupt 0 //中断程序
{
    key=KeyScan(); //获取键值
}
```

其中，主函数中没有对键盘的扫描语句，因此可以用来执行其他用户程序。键盘扫描程序放置在中断服务例程 FuncIr 中。这里的键盘扫描程序，可以采用前面介绍的扫描法来实现。

23.4 矩阵式键盘的接口实例

前面介绍了矩阵式键盘的结构，以及单片机系统中经常采用的扫描法、线反转法和中断法。实际的单片机系统中，采用线反转法比较多。下面给出一个完整电路实例，其中采用线反转法来实现单片机对矩阵式键盘的识别。单片机根据键值的大小，使 LED 闪烁相应的次数。

23.4.1 电路图

系统完整的电路图，如图 23-8 所示。这里的单片机选用 Atmel 公司的新型单片机 AT89S52，也可以采用其他兼容的 51 系列单片机，如 AT89S51、AT89C51、8051 等。该电路所需的元器件如表 23-1 所示。

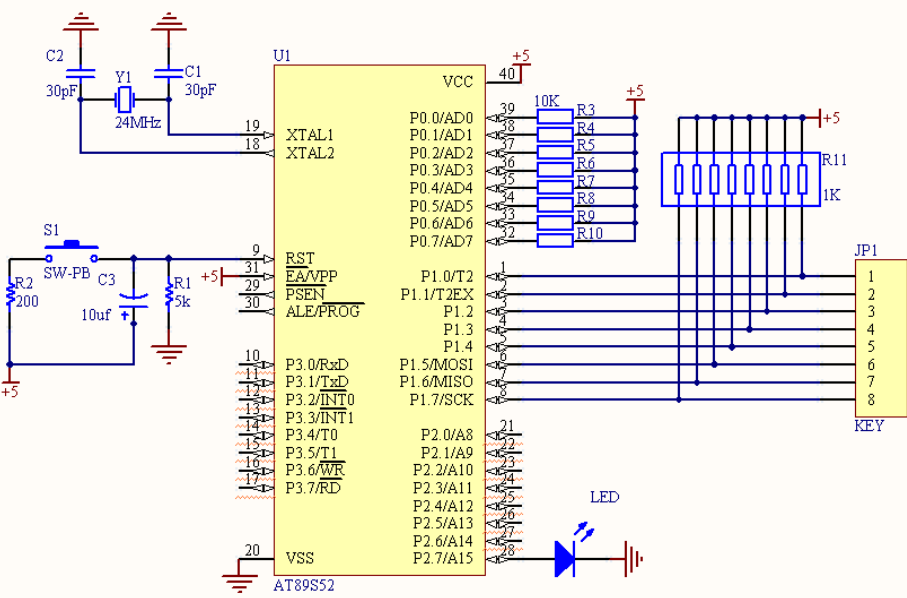


图 23-8 电路图

表 23-1 元器件列表

元 件	数 值	元 件	数 值
U1	AT89S52	R3~R10	10kΩ
C1、C2	30pF	R11	1kΩ 排阻

Y1	24MHz	JP1	4×4 矩阵式键盘
R1	5k Ω	LED	发光二极管
R2	200 Ω	S1	复位开关

其中 KEY 为 4×4 矩阵式键盘，R11 为键盘上拉电阻排，R3~R10 为单片机 P0 口的上拉电阻。P2.7 端口外接一个发光二极管，用于根据按键的键值来闪烁相应的次数。

4×4 矩阵式键盘的列分别接 P1.0~P1.3，行分别接 P1.4~P1.7。行号和列号的定义，如表 23-2 所示。

表 23-2 行号和列号的定义

	编 号	引 脚
行 号	1	P1.7
	2	P1.6
	3	P1.5
	4	P1.4
列 号	1	P1.3
	2	P1.2
	3	P1.1
	4	P1.0

23.4.2 程序设计

本例的程序功能是采用线反转法来扫描查询 4×4 矩阵式键盘，如果检测到按键按下，则闪烁发光二极管，否则将熄灭发光二极管。

1. 创建项目

这里采用 Keil C51 语言编写程序。具体操作步骤如下：

(1) 首先在 Keil μ Vision3 集成开发环境中，选择“Project”→“New”→“μ Vision Project”命令，新建一个工程，并保存。

(2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 23-9 所示。

(3) 单击“确定”按钮，此时弹出“μ Vision3”对话框，如图 23-10 所示。单击“是”按钮，完成工程的建立。

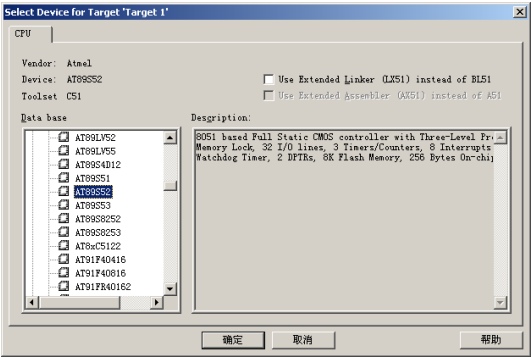


图 23-9 选择单片机 AT89S52



图 23-10 “μ Vision3”对话框

(4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件。可以在其中输入程序代码。

2. 程序代码

程序中主函数执行循环键盘扫描，由 while 循环构成。通过调用 KeyScan 函数来获得键值，并根据键值来循环点亮熄灭发光二极管。主程序的流程图，如图 23-11 所示。主程序代码示例

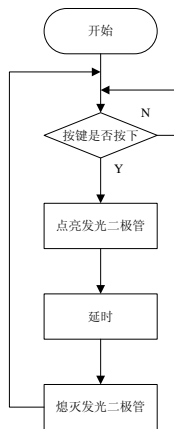


图 23-11 流程图

如下：

```
#include<stdio.h> //头文件
#include<reg52.h> //头文件

int row; //定义行号
int tier; //定义列号

int KeyScan(); //子函数声明
void Delay();

void main() //主函数
{
    int Key,i;
    while(1) //开始循环扫描键盘
    {
        Key=KeyScan(); //键盘扫描子函数
```

```
        for(i=0;i<Key;i++) //根据键值，使 LED 闪烁相应的次数
        {
            P2=0x80; //点亮发光二极管
            Delay();
            P2=0x00; //熄灭发光二极管
            Delay();
        }
    }
}
```

另外，该程序还分别自定义了两个函数，分别用于延时和线反转法键盘扫描。对于键盘扫描部分，程序中采用了软件去抖动的方法来消除按键抖动的影响。程序代码示例如下：

```
void Delay() //延时子函数
{
    long i;
    for(i=10000;i>0;i--); //可以根据系统的时钟频率来调整
}

int KeyScan() //键盘扫描子函数
{
    P1=0xF0; //列输出全 0
    if((P1&0xF0)!=0xF0) //扫描行，如果不全为 0，则进入
    {
        switch(P1) //获得行号
        {
            case 0x70:
                row=1;
                break;
            case 0xB0:
                row=2;
                break;
            case 0xD0:
                row=3;
                break;
            case 0xE0:
                row=4;
                break;
            default:
                break;
        }
    }
}
```



```
Delay(); //延时去抖动
P1=0x0F; //行输出全 0
if ((P1&0x0F)!=0x0F) //扫描列，如果不全为 0，则确认按键按下
{
    switch(P1) //获得列号
    {
        case 0x07:
            tier=1;
            break;
        case 0x0B:
            tier=2;
            break;
        case 0x0D:
            tier=3;
            break;
        case 0x0E:
            tier=4;
            break;
        default:
            break;
    }
    return 4*(row-1)+tier; //扫描到按键，返回键值
}
else
{
    return 0; //未扫描到按键按下，返回 0
}
```

23.4.3 程序仿真

在程序设计完成后，采用软件仿真可以模拟程序运行，从而及时发现问题。Keil μ Vision3 集成开发环境中没有矩阵式键盘的仿真模型，但在前面介绍过，4×4 矩阵式键盘的按键都对应一个行号和列号。当按键按下时，对应的两次线反转扫描，其中的行号或列号会输出低电平。为此，模拟矩阵式键盘的某个按键按下，则只需在列扫描的时候，将对应的行号输入低电平；在行扫描的时候，将对应的列号输入低电平即可。

软件中不能模拟实际硬件按键的抖动情况，但可以仿真整个程序运行的流程。首先进行仿真准备工作，具体操作如下：

- (1) 选择“Debug”→“Start/Stop Debug Session”命令，进入仿真调试模式。
- (2) 选择“Peripherals”→“I/O-Ports”→“Port 1”命令，打开并行口 1 的仿真面板，如图 23-12 所示。同样的操作可以打开并行口 2 的仿真面板，如图 23-13 所示。
- (3) 在源代码编辑窗口中，右键单击变量 row，选择“Add ‘row’ to Watch Window”→“#1”命令，将全局变量 row 添加到变量观察窗口 1。以同样的方法，将 tier 变量添加到变量观察窗口 1，如图 23-14 所示。

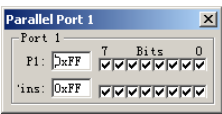


图 23-12 并行口 1 仿真面板



图 23-13 并行口 2 仿真面板

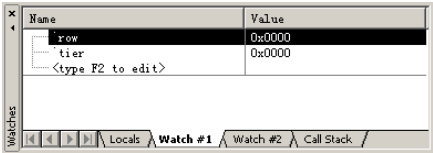


图 23-14 变量观察窗口

下面开始仿真操作，这里假定被按下的按键的行号为 2，列号为 3。具体操作步骤如下：

(1) 首先按 F11 键，单步执行到扫描行的地方。按照前面的表 23-2 的定义，行号为 2 的输入引脚是 P1.6。因此，将并口 1 仿真面板的 P1.6 端口设置为低电平，如图 23-15 所示。

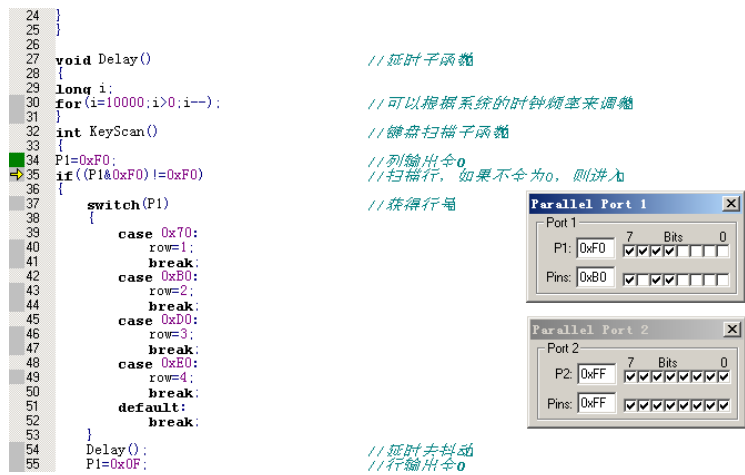


图 23-15 按键的行号输入

(2) 按 F11 键, 单步向下执行, 在 switch 语句中, 扫描获得行号, 可以在“Watch #1”窗口中观察到变量 row 的值, 如图 23-16 所示。

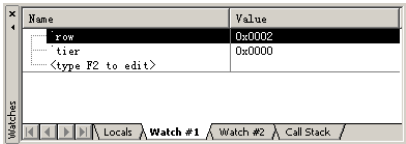


图 23-16 “Watch #1”窗口

- (3) 按 F11 键, 继续单步执行, 跳过 Delay 函数。
- (4) 执行到列扫描的地方, 此时按照前面的表 23-2 的定义, 列号为 3 的输入引脚是 P1.1。因此, 将并口 1 仿真面板的 P1.1 端口设置为低电平, 如图 23-17 所示。

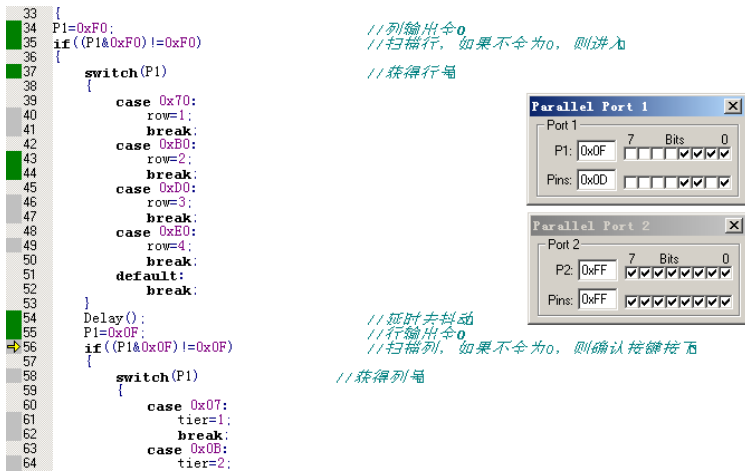


图 23-17 按键的列号输入

- (5) 按 F11 键, 单步向下执行, 在 switch 语句中, 扫描获得列号, 可以在“Watch #1”窗口中观察到变量 tier 的值, 如图 23-18 所示。
- (6) 按 F11 键, 继续执行。KeyScan 函数返回按键的键值, 程序根据键值的大小, 循环置 P2.7 端口为 0 或 1, 相当于点亮熄灭发光二极管, 如图 23-19 所示。

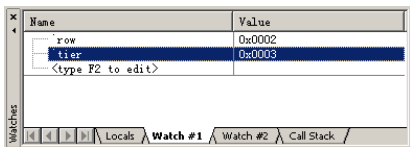


图 23-18 “Watch #1” 窗口

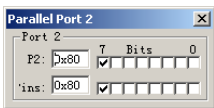


图 23-19 并行端口 2 的输出

至此，整个程序的仿真流程已经进行完毕。程序完全符合预期的工作流程，可以保证程序能够在硬件上顺利执行。

23.5 小结

本章详细讲述了键盘设计需要注意的一些问题，然后介绍了独立式按键和矩阵式键盘的工作方式。其中着重讲了矩阵式键盘的扫描法、线反转法和中断法的工作原理及程序设计。最后通过一个完整电路，实现了矩阵式键盘的扫描查询方式在程序中的应用。在实例中，还通过仿真分析了整个程序流程的正确性。矩阵式键盘应用十分广泛，熟练掌握它的使用是学习单片机应用系统的基础。

第24章 LED 数码管显示

对于人机交互式单片机系统来说，不仅需要响应用户输入，同时也需要将一些测控信息输出显示。这些显示信息可以提供实时的数据或图形结果，以便于掌握系统的状态并进行分析处理。目前，在单片机系统中最常用的是 LED 数码管显示。其成本低廉、使用简便，可以显示数字或几个特定的字符。本章将主要介绍 LED 数码管的种类与结构，以及使用 51 系列单片机如何实现数码管显示，包括静态显示和动态显示。本章还将通过实例介绍单个 LED 和多个 LED 的使用方法。

24.1 LED 数码管概述

LED 即发光二极管，英文全称为 Light Emitting Diode。单独的发光二极管便是一个最简单的 LED，通过控制其的亮灭来作为信号指示，一般用于电源指示灯、工作状态指示等。单个的发光二极管使用比较简单。

LED 数码管是由若干个发光二极管组成的显示字段的显示器件，一般简称为数码管。当数码管中的某个发光二极管导通的时候，相应的一个字段便发光，不导通的则不发光。LED 数码管可以根据控制不同组合的二极管导通，来显示各种数据和字符。

单片机应用系统中使用最多的是 7 段 LED，其可以显示十进制数字及一些英文字符。7 段 LED 显示模块可以分为共阴极和共阳极两种，下面分别进行介绍。

24.1.1 7 段共阳极 LED 结构及显示段码

7 段共阳极 LED 数码管是由 7 个条形发光二极管和一个小数点位构成，其引脚配置，如图 24-1 所示，其内部结构，如图 24-2 所示。从图 24-1 中可以看出，其中 7 个发光二极管构成字形“8”，可以用来显示数字，另一个发光二极管构成小数点。因此，这种数码管有时也被称为 8 段 LED 数码管显示器。

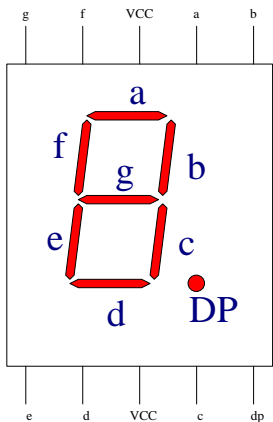


图 24-1 7 段共阳极 LED 引脚配置

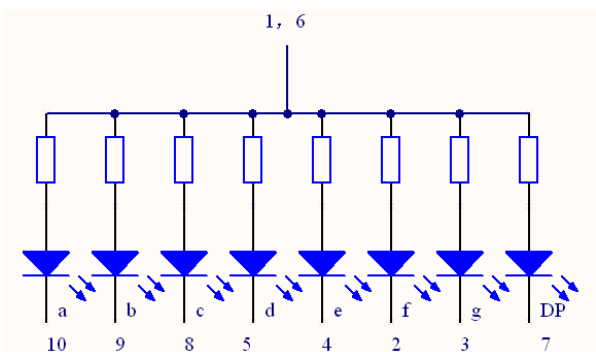


图 24-2 7 段共阳极 LED 内部结构

在 7 段共阳极 LED 数码管中，发光二极管的阳极为公共端，接高电平+5V，当某个发光二

极管的阴极为低电平的时候，发光二极管导通，该字段发光；反之，如果某个发光二极管的阴极接高电平的时候，发光二极管截止，则该字段不发光。

由于 7 段 LED 数码管加上小数点位 DP，共有 8 个发光单元，正好组合成一个整 8 位字节。这种结构使其和单片机的接口十分方便，可以直接将 8 个显示字段引脚接到单片机的一个 8 位并行 I/O 端口上。

7 段共阳极 LED 数码管和字节的对应关系，如图 24-3 所示。这样便可以直接从单片机并口输出数据，控制字段的亮灭来显示不同的数据和字符等。对于 7 段共阳极 LED 数码管显示字符和单片机并口输出数据的关系，如表 24-1 所示。

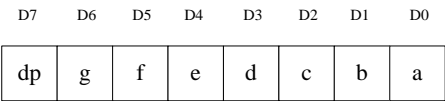


图 24-3 LED 数码管的字节对应关系

表 24-1 7 段共阳极 LED 显示字符及段码

显示字符	共阳极 LED 段码	显示字符	共阳极 LED 段码
0	C0H	C	C6H
1	F9H	D	A1H
2	A4H	E	86H
3	B0H	F	8EH
4	99H	P	8CH
5	92H	U	C1H
6	82H	Γ	CEH
7	F8H	Y	91H
8	80H	H	89H
9	90H	L	C7H
A	88H	全亮	00H
B	83H	全灭	FFH

如果采用汇编语言进行程序设计，则可以使用 MOV 指令来向端口写数据，从而实现控制 LED 数码管的显示，示例如下：

```
MOV P1,#0A4H
```

该指令写共阳极 LED 段码 A4H，则 LED 数码管显示数字 2。如果采用 C51 语言进行程序设计，则可以直接为端口寄存器赋值，示例如下：

```
P1=0xA4;
```

24.1.2 7 段共阴极 LED 结构及显示段码

7 段共阴极 LED 数码管和共阳极 LED 数码管结构类似，其引脚配置，如图 24-4 所示。从图 24-4 中可以看出 7 段共阴极 LED 数码管同样由 8 个发光二极管组成，其中 7 个发光二极管构成字形“8”，另一个发光二极管构成小数点。

共阴极 7 段 LED 数码管的内部结构，如图 24-5 所示。其中所有发光二极管的阴极为公共端，接 GND。如果发光二极管的阳极为高电平的时候，发光二极管导通，该字段发光；反之，如果发光二极管的阳极为低电平的时候，发光二极管截止，该字段不发光。

对于 7 段共阴极 LED 数码管显示字符和单片机并口输出数据的关系，如表 24-2 所示。这样便可以直接从单片机并口输出数据，控制字段的亮灭来显示不同的数据和字符等。

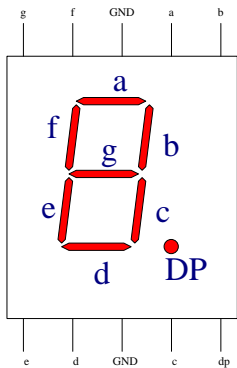


图 24-4 7 段共阴极 LED 引脚配置

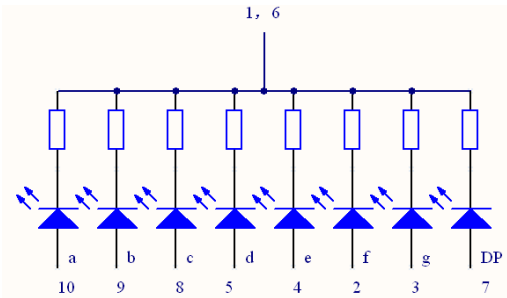


图 24-5 7 段共阴极 LED 内部结构

表 24-2 7 段共阴极 LED 显示字符码

显示字符	共阴极 LED 段码	显示字符	共阳极 LED 段码
0	3FH	C	39H
1	06H	D	5EH
2	5BH	E	79H
3	4FH	F	71H
4	66H	P	73H
5	6DH	U	3EH
6	7DH	Γ	31H
7	07H	Y	6EH
8	7FH	H	76H
9	6FH	L	38H
A	77H	全亮	FFH
B	7CH	全灭	00H

从表 24-2 中可以看出,7 段共阴极 LED 数码管和共阳极 LED 数码管的段码是互为补数的。如果采用汇编语言进行程序设计,则可以使用 MOV 指令来向端口写数据,从而实现控制 LED 数码管的显示,示例如下:

```
MOV P1,#79H
```

该指令写共阴极 LED 数码管段码 A4H,则 LED 数码管显示字符“E”。如果采用 C51 语言进行程序设计,则可以直接为端口寄存器赋值,示例如下:

```
P1=0x79;
```

24.2 单个 LED 驱动实例

前面介绍了 LED 数码管的结构及其显示方式。LED 数码管主要用于显示数字和一些特定的字符。下面通过一个具体的实例介绍如何使用 51 系列单片机进行数字和字母的显示。

24.2.1 电路图

本例主要用来使用共阳极 LED 数码管显示数字或字符,读者可以从中掌握 LED 数码管的基本操作方法。这里给出完整的电路原理图,如图 24-6 所示。

本电路中所使用的元器件列表,如表 24-3 所示。

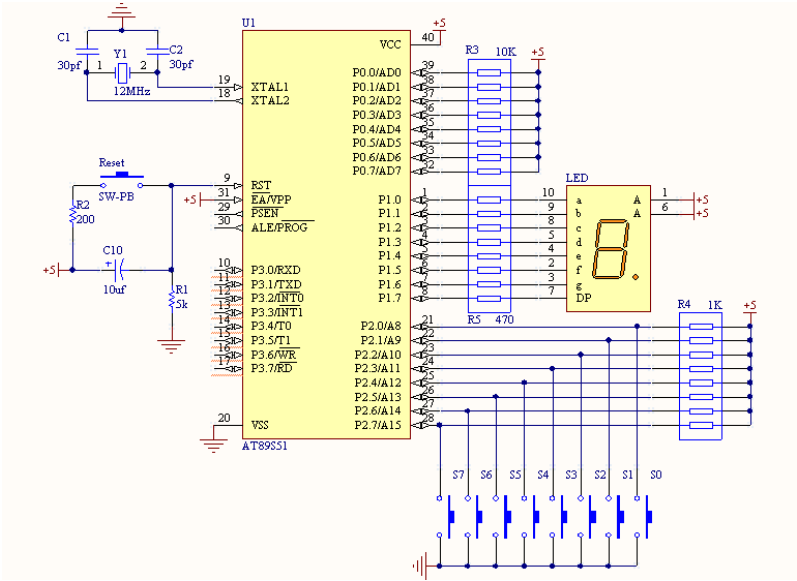


图 24-6 电路原理图

表 24-3 元件列表

元 器 件	数 值	数 量
U1	AT89S51	1 个
D1	共阳极 LED	1 个
C1、C2	30pF	2 个
C10	10μF 电解	1 个
S0~S7、Reset	按键开关	9 个
Y1	12.0000MHz 晶振	1 个
R1	5KΩ	1 个
R2	200Ω	1 个
R3	10KΩ上拉排阻	1 个
R4	1KΩ上拉排阻	1 个
R5	470Ω排阻	1 个

在本电路中，共阳极 LED 数码管的公共端接 5V，其余分别与 P1 口通过一个限流电阻相连。当 P1 端口某个引脚输出低电平，则该段发光；如果输出高电平，则该段不发光。下面分别介绍电路的各个部分的组成及功能。

1. 振荡电路

由于 AT89S51 内部含有一个振荡器，可以直接作为 CPU 的时钟源，也可以由外部振荡器输入时钟信号，作为 CPU 的时钟源。这里采用片内振荡器的工作方式。在 XTAL1 和 XTAL2 之间连接一个 12MHz 的石英晶体及两个陶瓷电容，构成并联谐振电路。

2. 复位电路

单片机的硬件复位基本点是在 RST 引脚输入两个以上机器周期的高电平，便可以实现硬件复位。本电路所采用的复位电路，既可以上电复位，也可以手动按复位按键复位。图 24-6 中的电阻、电容值为参考值。一般电阻、电容的取值随时钟频率的不同而变化，读者可以通过实验选取合适的数值。

3. 数码管接口

单片机的 P1 端口通过 470Ω 的限流电阻与 8 段共阳极数码管连接，数码管的公共端连接 5V

51 单片机开发与应用技术详解

电源的正极。P1 端口各个引脚输出低电平时，发光二极管导通，LED 发光；P1 端口各个引脚输出高电平，发光二极管截止，LED 不发光。

4. 开关输入接口

单片机 P2 口分别接 8 个开关电路，当按键开关断开时，端口输入高电平；当案件开关闭合时，端口输入低电平。单片机程序扫描端口，根据端口的输入电平，决定如何显示数码管。

此外，单片机的 P0 口接 10kΩ的上拉电阻， \overline{EA}/VPP 引脚接高电平，表示采用内部程序存储器。

24.2.2 程序设计

这里采用 LED 的静态显示的方式，根据按键的不同，使 LED 数码管有不同的显示输出。

1. 创建项目

这里采用 Keil C51 语言编写程序。具体操作步骤如下：

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S51，如图 24-7 所示。
- (3) 单击“确定”按钮，此时弹出“μ Vision3”对话框，如图 24-8 所示。单击“是”按钮，完成工程的建立。

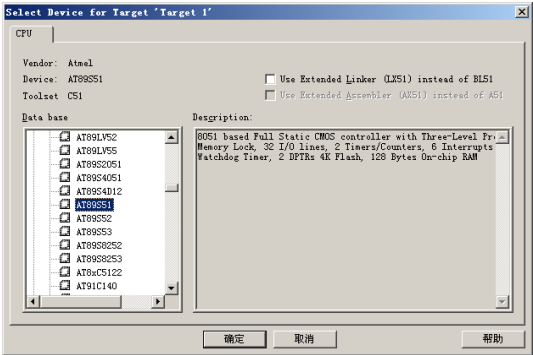


图 24-7 选择单片机 AT89S51

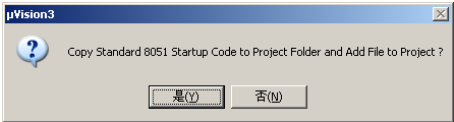


图 24-8 “μ Vision3”对话框

- (4) 选择“File”→“New”命令，新建一个程序文件，并保存为 *.C 文件。可以在其中输入程序代码。

2. 程序代码

程序中主函数的流程图，如图 24-9 所示。AT89S51 上电时，首先使 LED 数码管全部亮灭两次，表示进入工作状态。在主循环中，根据扫描按键的结果来使 LED 数码管显示不同的数字和字符。

主程序示例如下：

```
#include <reg51.h> //头文件
#include <intrins.h>

void Delay(); //子函数声明

void LEDShow(); //扫描显示字程序

void main() //主函数
{
    P1=0x00; //LED 全亮
```

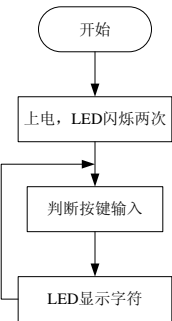


图 24-9 主程序流程图


```

Delay(); //延时
P1=0xFF; //LED 全灭
Delay(); //延时
P1=0x00; //LED 全亮
Delay(); //延时
P1=0xFF; //LED 全灭
Delay();
while(1) //循环
{
    LEDShow(); //扫描按键，显示数字或字符
}

```

其中，用到两个自定义函数 Delay 和 LEDShow。下面分别进行介绍。

➤ 延时函数 Delay。该函数中调用_nop_函数来实现延时，其位于头文件 intrins.h 中。该函数的示例代码如下：

```

void Delay() //延时子函数
{
    int i;
    for(i=0;i<1000;i++) //循环
    {
        _nop_(); // _nop_ 函数延时
    }
}

```

➤ 扫描显示函数 LEDShow。在该函数中，通过 if 语句对按键进行判断，根据判断的结果，使 P1 端口输出相应的共阳极 LED 数码管的显示段码。该函数的示例代码如下：

```

void LEDShow()
{
    if (P2==0xFE) //S0 按下
    {
        P1=0xC0; //显示字符“0”
    }
    else if (P2==0xFD) //S1 按下
    {
        P1=0xF9; //显示字符“1”
    }
    else if (P2==0xFB) //S2 按下
    {
        P1=0xA4; //显示字符“2”
    }
    else if (P2==0xF7) //S3 按下
    {
        P1=0xB0; //显示字符“3”
    }
    else if (P2==0xEF) //S4 按下
    {
        P1=0x99; //显示字符“4”
    }
    else if (P2==0xDF) //S5 按下
    {
        P1=0x92; //显示字符“5”
    }
}

```

```
else if(P2==0xBF)           //S6 按下
{
    P1=0x82;                 //显示字符“6”
}
else if(P2==0x7F)           //S7 按下
{
    P1=0xF8;                 //显示字符“7”
}
else if(P2==0xFC)           //S2、S1 同时按下
{
    P1=0x80;                 //显示字符“8”
}
else if(P2==0xFA)           //S2、S1 同时按下
{
    P1=0x90;                 //显示字符“9”
}
else if(P2==0xF6)           //S3、S1 同时按下
{
    P1=0x88;                 //显示字符“A”
}
else if(P2==0xEE)           //S4、S1 同时按下
{
    P1=0x83;                 //显示字符“B”
}
else if(P2==0xDE)           //S5、S1 同时按下
{
    P1=0xC6;                 //显示字符“C”
}
else if(P2==0xBE)           //S6、S1 同时按下
{
    P1=0xA1;                 //显示字符“D”
}
else if(P2==0x7E)           //S7、S1 同时按下
{
    P1=0x86;                 //显示字符“E”
}
else
{
    P1=0xFF;                 //LED 不显示
}
}
```

程序编写完毕后，便可以进行程序编译仿真。通过仿真操作可以观察各个引脚电平的执行，以及按键对应的不同 LED 显示字符。

当仿真的时序没有问题的時候，便可以将程序下载到单片机中执行。

24.3 多个 LED 驱动方式

在实际的单片机应用系统中，使用单个 LED 数码管的情况比较少，经常需要同时使用多个 LED 数码管来显示大于一位的数据或字符串。以 4 个 LED 数码管并列使用的情况为例，如图 24-10 所示。这 4 个数码管可以显示-999~9999 之间的任何数字，也可以同时显示 4 个字符构成的字符串。可见使用多个 LED 数码管可以大大扩展显示的信息量。

对于使用单个 LED 数码管的场合，直接用单片机的一个并行口便可以控制显示。如果仍然采用这种方法来控制显示 N 个 LED 数码管显然是不太可能的，因为典型的 8051 单片机只有 4 个 I/O 并口，而且有些 I/O 口还需要用做其他用途。而对于一些多引脚的型号，通常也不够为每个 LED 分配一个 I/O 并口用于显示。此时便需要根据系统资源占用情况，来选用合理的显示控制方式。

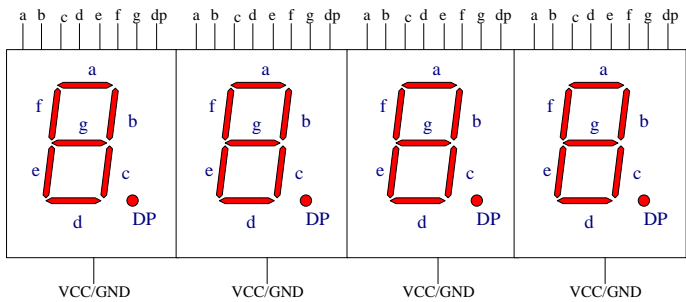


图 24-10 多个 LED 数码管并列使用

对于多个 LED 数码管并用的场合，一般有静态显示、动态显示和 LED 驱动器三种显示驱动方式。

24.3.1 静态驱动显示

LED 数码管静态显示方式是指当数码管显示某个字符的时候，相应字段的发光二极管恒定地导通或截止，即亮灭是完全不变的。在这种情况下，多个 LED 是同时显示的。

这里以 4 个共阴极 LED 数码管为例，如图 24-11 所示。其公共端接 GND，每个 LED 数码管的字段引脚分别接单片机的 P0、P1、P2、P3 端口，这样便可以为每个数码管单独进行赋值操作。

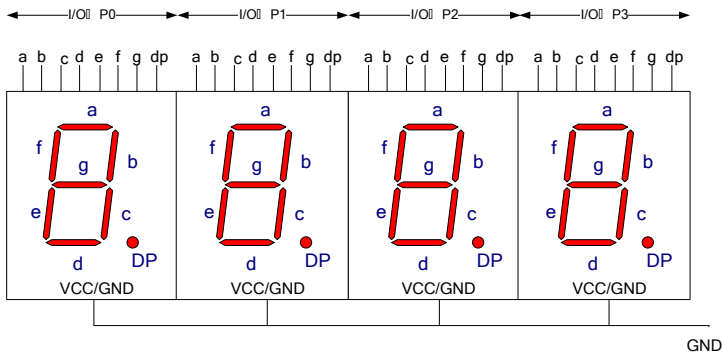


图 24-11 4 个 LED 并列使用的原理图

这种显示方式的优点是，接口操作简单，只需将显示字符相应的字段码发送到 LED，并在端口保持即可；静态显示字符时，只需较小的驱动电流便可以获得较高的显示亮度。其缺点是当 LED 数量比较多时，需要很多的 I/O 线，对硬件资源的要求比较苛刻。

对于上面的控制显示方式，显然整个系统基本上没有引脚资源用于其他操作了。因此，一般不能直接使用单片机有限的并行 I/O 接口，实际应用中往往采用外部扩展并行接口的方法。

外部扩展并行 I/O 接口，可以通过串行接口和使用外部扩展 RAM 地址两种方法。下面分别进行介绍。

1. 通过串行接口扩展 LED 显示

通过串行接口扩展 LED 显示的原理是基于串行接口模式 0 的外部输出端口扩展。通过外

51 单片机开发与应用技术详解

接串入/并出的移位寄存器，可以扩展出多个 8 位并行 I/O 接口，如图 24-12 所示。其中使用了 4 个串入/并出移位寄存器 CD4094，STB 为数据输出允许控制端，当 STB=1 时，将接收到的串行数据转换成并行数据输出；当 STB=0 时，CD4094 输出保持不变，对接收到的数据不处理。

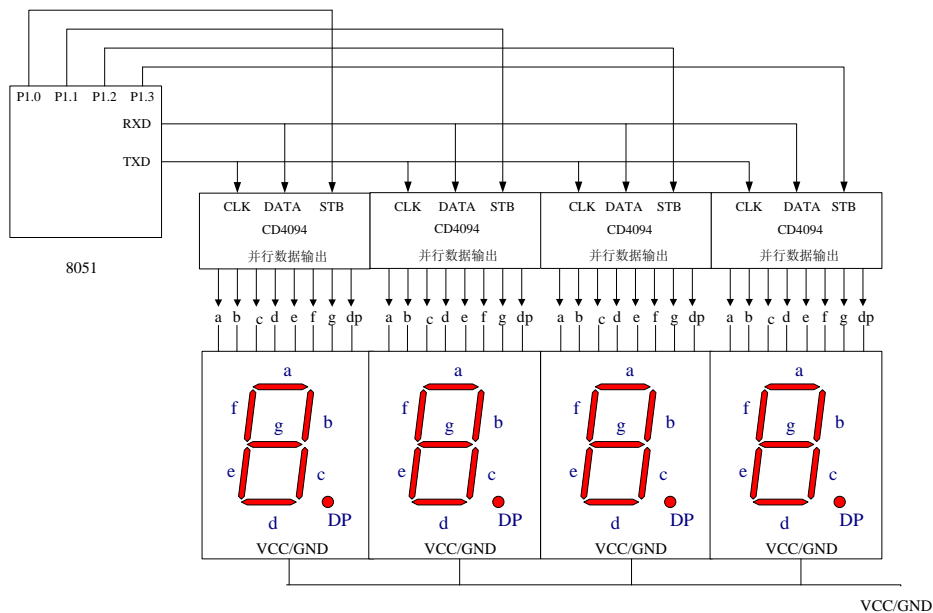


图 24-12 串行接口扩展 LED 显示原理图

如果采用汇编语言进行程序设计，其程序示例如下：

```
ORG      0200H
MAIN:    MOV     SCON,    #00H           ; 串行口模式 0 的初始化
        CLR     ES              ; 禁止串行中断
        CLR     P1.0           ; 关闭 LED0 的输出
        CLR     P1.1           ; 关闭 LED1 的输出
        CLR     P1.2           ; 关闭 LED2 的输出
        CLR     P1.3           ; 关闭 LED3 的输出
        MOV     A, #0C0H       ; 字符 0 的段码送入累加器 A
        MOV     SBUF, A        ; 输出字符 0
        JNB     TI, $          ; 查询 TI, 若 TI=0, 未发送完毕, 等待
        SETB    P1.0           ; TI=1, 发送完毕, 启动并行输出
        CLR     TI             ; 清零 TI
        CLR     P1.0           ; 置 STB=0
        MOV     A, #0F9H       ; 字符 1 的段码送入累加器 A
        MOV     SBUF, A        ; 输出字符 1
        JNB     TI, $          ; 查询 TI, 若 TI=0, 未发送完毕, 等待
        SETB    P1.1           ; TI=1, 发送完毕, 启动并行输出
        CLR     TI             ; 清零 TI
        CLR     P1.1           ; 置 STB=0
        MOV     A, #0A4H       ; 字符 2 的段码送入累加器 A
        MOV     SBUF, A        ; 输出字符 2
```

```

JNB      TI,$           ;查询 TI, 若 TI=0, 未发送完毕, 等待
SETB     P1.2           ;TI=1, 发送完毕, 启动并行输出
CLR      TI             ;清零 TI
CLR      P1.2           ;置 STB=0
MOV       A,#0B0H       ;字符 3 的段码送入累加器 A
MOV      SBUF,A         ;输出字符 3
JNB      TI,$           ;查询 TI, 若 TI=0, 未发送完毕, 等待
SETB     P1.3           ;TI=1, 发送完毕, 启动并行输出
CLR      TI             ;清零 TI
CLR      P1.3           ;置 STB=0
END

```

在该程序中, 首先初始化串行口, 并置 P1.0~P1.3 均为 0, 所有的 CD4094 均不输出。接着发送 LED0 的数据至 SBUF, 然后置 P1.0=1, 则扩展 I/O 口 0 便输出并行数据驱动 LED 数码管, 然后依次写 LED1、LED2 和 LED3, 最终完成显示。如果采用 C51 语言进行程序设计, 其程序示例如下:

```

#include "reg51.h"           //头文件

sbit STB0=P1^0;             //位定义
sbit STB1=P1^1;
sbit STB2=P1^2;
sbit STB3=P1^3;

void main (void)            //主函数
{
    SCON=0x00;              //初始化串口
    ES=0;
    STB0=0;                 //禁止 CD4094 输出
    STB1=0;
    STB2=0;
    STB3=0;

    SBUF=0xC0;              //发送 LED0 数据
    while (TI==0)
    {
        STB0=1;
        TI=0;
        STB0=0;

        SBUF=0xF9;          //发送 LED1 数据
        while (TI==0)
        {
            STB1=1;
            TI=0;
            STB1=0;

            SBUF=0xA4;       //发送 LED2 数据
            while (TI==0)
            {
                STB2=1;
                TI=0;
                STB2=0;
            }
        }
    }
}

```

```
SBUF=0xB0; //发送 LED3 数据
while(TI==0)
{
}
STB3=1;
TI=0;
STB3=0;

while(1) //主循环
{
}
}
```

在该程序中，定义了 CD4094 的控制端口 STB0~STB3。在主函数中，首先初始化串行口，并置 STB0~STB3 均为 0，所有的 CD4094 均不输出。接着发送 LED0 的数据至 SBUF，然后置 STB0=1，则扩展 I/O 口 0 便输出并行数据驱动 LED 数码管，然后依次写 LED1、LED2 和 LED3，最终完成显示。

上面的程序均可以在 Keil μ Vision3 编译环境中执行，并可以下载到前面介绍的 AT89S52 最小系统中运行。

2. 通过外部 RAM 地址空间扩展 LED 显示

通过外部 RAM 地址空间扩展 LED 显示的原理是使用片外 RAM 地址作为扩展并行 I/O 输出接口，用来控制显示 LED 数码管。其原理图如图 24-13 所示。单片机片外有 64KB 的 RAM 空间，因此可以扩展多个 I/O 接口用于 LED 数码管的显示。

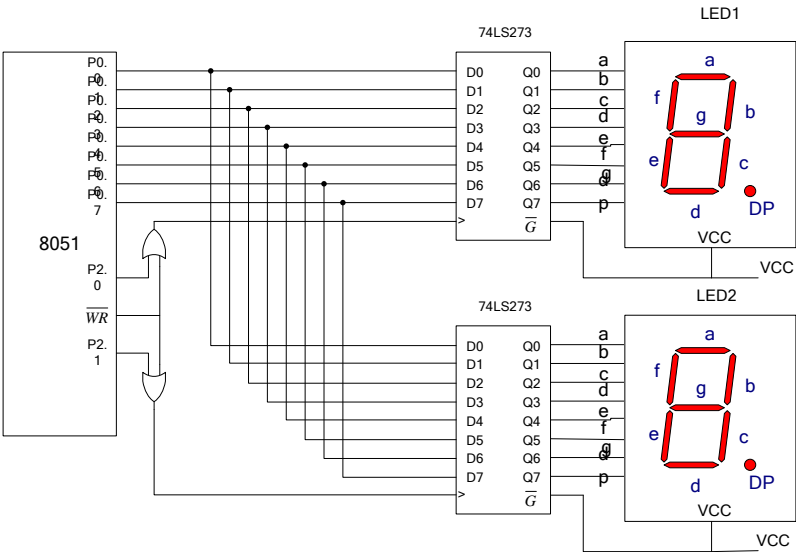


图 24-13 外部 RAM 地址空间扩展 LED 显示原理图

根据前面章节介绍的关于并行 I/O 扩展的相关内容。这里采用 74LS273 作为扩展输出，使用的两个扩展地址分别为 FEFH 和 FDFH。P0 端口作为地址/数据复用。采用汇编语言进行程序设计，程序示例如下：

```
ORG 0400H
MAIN: MOV DPTR, #0FEFH ;数据指针指向扩展 LED1 的地址
      MOV A, #0F9H ;字符 1 的段码送入累加器 A
      MOV @DPTR, A ;向 LED1 输出显示
      MOV DPTR, #0FDFH ;数据指针指向扩展 LED2 的地址
```


51 单片机开发与应用技术详解

提供两个输出端口，PA 端口经过一个 8 位同相集成驱动芯片 BIC8718 驱动后接 LED 数码管的公共极，作为 LED 的位选信号。PB 口输出控制 LED 的段码，同样经过 BIC8718 芯片连接 LED 的各个段码引脚。

这里采用汇编语言进行程序设计，程序代码以子程序的形式给出。程序的流程图，如图 24-16 所示。程序代码示例如下：

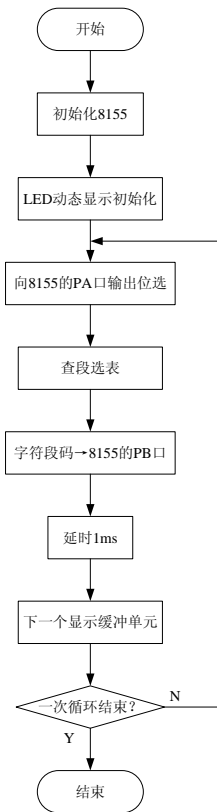


图 24-16 8 位 LED 动态显示流程图

DSHOW: MOV DPTR, #7F00H ; 8155 状态命令地址 7F00H

MOV A, #03H ; 初始化 8155 命令字

MOVX @DPTR, A ; 发送命令

MOV R4, #7FH ; 点亮第一个 LED

MOV R5, #78H

MOV A, R4

LED: MOV DPTR, #7F01H ; 指向 8155 的 PA 口

MOVX @DPTR, A ; LED 选择位送 PA 口

INC DPTR ; 指向 8155 的指向 PB 口

MOV A, @R5 ; 查 7 段代码

ADD A, #0DH

MOVC A, @A+PC

MOVX @DPTR, A ; 发送 7 段代码

ACALL DLY ; 延时

MOV A, R4

JNB A.0, LED1 ; 8 位显示完

RR A ; 指向下一个 LED 位


```
MOV      R4,A
INC      R5
AJPMP    LED0      ;转显示下一位
LED1:    SJMP    LED1
DSEG:    DB      3FH,06H,5BH,4FH,66H      ;7 段代码表
          6DH,7DH,07H,7FH,6FH
          77H,7CH,39H,5EH,79H
          71H,73H,3EH,31H,6EH
          FFH,00H
DLY:      MOV      R7,#02H      ;延时子程序
DL:        MOV      R6,#0FFH
DL1:      DJNZ     R6,DL1
          DJNZ     R7,DL
          RET
```

在该子程序中，使用 8051 单片机片内 RAM 的 78H~7FH 单元作为显示缓冲区，从低位到高位依次存放 8 个需要显示的数据。8155 的 PA 口每次输出只使 1 位为低电平，这样可使其中一个 LED 显示。8155 的 PB 口输出该 LED 需要显示数据的段码。程序中依次变化，可以实现动态 LED 的稳定显示。

24.3.3 LED 驱动器

前面介绍的 LED 的动态驱动显示在程序设计上比较复杂，实际上可以将相应的 LED 扫描动态显示电路交由特定功能的芯片来完成。目前，市场上有多种 LED 数码管显示驱动芯片，如 MAXIM 公司生产的 MAX7219 显示驱动器。

1. LED 驱动器 MAX7219 简介

MAX7219 是 7 段共阴极 LED 显示驱动器，采用三线串行方式与 8051 通信，电路结构十分简单。MAX7219 片内集成了 BCD 码到 B 码的译码器、多路复用扫描电路、LED 字段和字位驱动电路及 RAM 存储器。MAX7219 可以驱动 8 个 7 段共阴极 LED 显示器，通过一个 10KΩ 左右的外接电阻可以设置所有 LED 的段电流。MAX7219 具有低电压保持，只要外接电压超过 2V 便可以保存数据。

典型的 DIP 封装的 MAX7219，如图 24-17 所示。其各引脚的功能如下所述。

- DID 0~DID 7：8 个字位驱动引脚；
- SEG A~G，dp：7 段驱动输出；
- SEG dp：小数点驱动输出；
- CLK：时钟输入，最高时钟频率为 10MHz；
- DIN：串行数据输入。在 CLK 时钟的上升沿，串行数据被移入 MAX7219 内部移位寄存器，移入时最高位在前；
- DOUT：串行数据输出。输入到 DIN 的数据经过 16.5 个时钟周期后，在 DOUT 端有效。在 CLK 的下降沿数据移出；
- ISET：峰值段电流设置。可以通过一个 10KΩ 的上拉电阻 R_{SET} 来设置峰值段电流；
- LOAD：加载输入数据。LOAD 信号必须第 16 个上升沿同时或之后，但在下一个时钟上升沿之前变高，否则将会丢失数据；
- V+：+5V 外接电源；
- GND：接地，两个 GND 引脚必须相连。

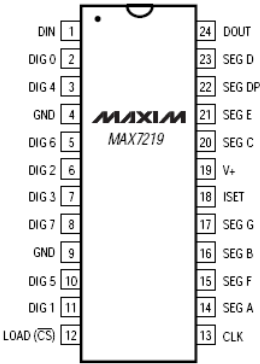


图 24-17 MAX7219 的引脚排列

2. MAX7219 数据传输方式

MAX7219 采用串行数据传输方式，串行数据以 16 位为一帧，如表 24-4 所示。其中 D7~

D0 为寄存器数据，D11~D8 为内部寄存器地址，D15~D12 可以任意。

表 24-4 MAX7219 的串行数据格式

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
-	-	-	-	内部寄存器地址				寄存器数据							

这 16 位的数据帧在每个 CLK 的上升沿，通过 DIN 引脚，被移入到内部 16 位移位寄存器中。然后在 LOAD 的上升沿将数据锁存到 MAX7219 片内数字或控制寄存器中。而 DOUT 端的数据在 CLK 的下降沿输出。MAX7219 的数据传输时序，如图 24-18 所示。

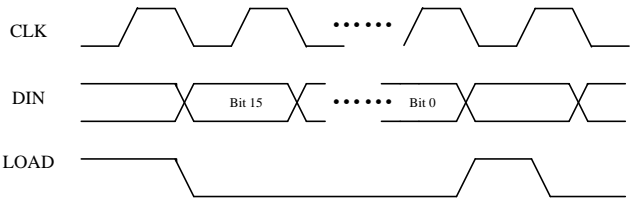


图 24-18 MAX7219 的数据传输时序

3. MAX7219 的内部寄存器

MAX7219 片内具有 14 个内部数字和控制寄存器，如表 24-5 所示。其中，控制寄存器有 5 个，分别是译码方式寄存器、显示亮度寄存器、扫描界限寄存器、停机寄存器和显示测试寄存器。数字寄存器共 8 个，由一个片内 8×8 双端口 SRAM 实现。另外还有一个 NO-OP 寄存器，用于多个 MAX7219 级联使用，可以在不改变显示或影响任意控制寄存器的条件下，数据从 DIN 传送到 DOUT。

表 24-5 MAX7219 内部寄存器及其地址

寄 存 器	地 址						十六进制代码
	D15~D12	D11	D10	D9	D8		
NO-OP 寄存器	×	0	0	0	0		×0H
数字 0	×	0	0	0	1		×1H
数字 1	×	0	0	1	0		×2H
数字 2	×	0	0	1	1		×3H
数字 3	×	0	1	0	0		×4H
数字 4	×	0	1	0	1		×5H
数字 5	×	0	1	1	0		×6H
数字 6	×	0	1	1	1		×7H
数字 7	×	1	0	0	0		×8H
译码方式寄存器	×	1	0	0	1		×9H
亮度寄存器	×	1	0	1	0		×AH
扫描界限寄存器	×	1	0	1	1		×BH
停机寄存器	×	1	1	0	0		×CH
显示测试寄存器	×	1	1	1	1		×FH

MAX7219 的译码方式寄存器中，每一位与一个数字位相对应，如果对对应位为逻辑高电平表示该位使用 B 码译码，而逻辑低电平则表示该位不译码，如表 24-6 所示。

表 24-6 译码方式寄存器

含 义	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
0~7 位不译码	0	0	0	0	0	0	0	0	00H
0 位译成 B 码，7~1 不译码	0	0	0	0	0	0	0	1	01H

.....
0~3 位使用 B 码译码, 4~7 不译码	0 0 0 0 1 1 1 1	0FH
.....
0~7 位使用 B 码译码	1 1 1 1 1 1 1 1	FFH

MAX7219 的亮度寄存器用于调节 LED 的显示亮度。实际电路中, 在 ISET 和电源正极之间连接外部电阻 R 来控制显示亮度。R 既可为固定电阻, 也可为可变电阻, 其最小值为 9.25 KΩ。亮度寄存器中的数值表示了亮度的大小, 共 16 级亮度, 如表 24-7 所示。

表 24-7 亮度寄存器

亮 度	D7 D6 D5 D4 D3 D2 D1 D0	十六进制代码
1/32 (最小亮度)	× × × × 0 0 0 0	×0H
3/32	× × × × 0 0 0 1	×1H
5/32	× × × × 0 0 1 0	×2H
.....
29/32	× × × × 1 1 1 0	×EH
31/32 (最大亮度)	× × × × 1 1 1 1	×FH

MAX7219 的扫描界限寄存器用于设置需要显示的数字位, 其取值范围为 1~8。扫描界限寄存器数据的含义, 如表 24-8 所示。

表 24-8 扫描界限寄存器

显示数字位	D7 D6 D5 D4 D3 D2 D1 D0	十六进制代码
第 0 位数字显示	× × × × × 0 0 0	×0H
第 0~1 位数字显示	× × × × × 0 0 1	×1H
第 0~2 位数字显示	× × × × × 0 1 0	×2H
.....
第 0~6 位数字显示	× × × × × 0 1 1	×6H
第 0~7 位数字显示	× × × × × 1 1 1	×7H

一般来说, 显示的位数对 LED 数码管的亮度也有影响。当显示位数少的时候, 相同的亮度寄存器设置也会造成亮度的提高。因此, 设置 LED 数码管的亮度应该同时考虑 ISET 和电源之间的电阻、亮度寄存器值及扫描界限寄存器值。

MAX7219 的停机寄存器用于停止 LED 显示。当 MAX7219 处于停机工作方式时, 扫描振荡器停止工作, LED 所有的段都截止, 此时 LED 不显示任何数据, 而保存在各个寄存器中的数据将保持不变。MAX7219 的停机寄存器的含义, 如表 24-9 所示。

表 24-9 停机寄存器

工作方式	D7 D6 D5 D4 D3 D2 D1 D0	十六进制代码
停机工作	× × × × × × × 0	×0H
正常工作	× × × × × × × 1	×1H

MAX7219 的显示测试寄存器用于测试 LED 的好坏。其有两种工作方式, 即正常工作和显示测试。正常工作模式即一般的扫描显示模式; 而在显示测试方式下, 8 位 LED 数码管将按占空比 31/32 扫描, 而此时所有控制寄存器和数据寄存器中的值不起作用。MAX7219 显示测试寄存器的含义, 如表 24-10 所示。

表 24-10 显示测试寄存器

工作方式	D7 D6 D5 D4 D3 D2 D1 D0	十六进制代码
正常工作	× × × × × × × 0	×0H
显示测试	× × × × × × × 1	×1H

MAX7219 的数字寄存器用于设置 LED 数码管的显示数字。8 个数字寄存器由一个片内 8×8 双端口 SRAM 实现, 可以直接寻址, 因此可以对单个数字进行更新。

MAX7219 的数字寄存器受译码方式寄存器的控制，可以选择 B 译码或不译码。如果不译码，则数字寄存器中数据的 D0~D6 位分别对应 7 段 LED 显示器的 A~G 段，D7 位对应 LED 的小数点位 DP。某位数据为 1 则点亮与该位对应的 LED 段，而如果数据为 0 则该段熄灭。如果使用 B 译码，数字寄存器可将 BCD 码译成 B 码（0~9、-、E、H、L、P），如表 24-11 所示。

表 24-11 数字 0~7 寄存器

显示字符	寄存器数据						点 亮 段						
	D7	D6~D4	D3	D2	D1	D0	DP	A	B	C	D	E	F G
0	×	0	0	0	0	0	1	1	1	1	1	1	0
1	×	0	0	0	0	1	0	1	1	0	0	0	0
2	×	0	0	1	0	0	1	1	0	1	1	0	1
3	×	0	0	1	1	1	1	1	1	1	0	0	1
4	×	0	1	0	0	0	0	1	1	0	0	1	1
5	×	0	1	0	1	1	1	0	1	1	0	1	1
6	×	0	1	1	0	0	1	0	1	1	1	1	1
7	×	0	1	1	1	1	1	1	1	0	0	0	0
8	×	1	0	0	0	0	1	1	1	1	1	1	1
9	×	1	0	0	0	1	1	1	1	1	0	1	1
-	×	1	0	1	0	0	0	0	0	0	0	0	1
E	×	1	0	1	1	1	1	0	0	1	1	1	1
H	×	1	1	0	0	0	0	1	1	0	1	1	1
L	×	1	1	0	1	1	0	0	0	1	1	1	0
P	×	1	1	1	0	0	1	1	0	0	1	1	1
暗	×	1	1	1	1	1	0	0	0	0	0	0	0

其中，小数点位 DP 由 D7 位控制。当 D7=0 时，熄灭小数点；当 D7=1 时，则点亮小数点。

4. MAX7219 的级联

MAX7219 的级联是指多个 MAX7219 一起使用。实际电路中，可以将所有级联器件的 LOAD 端连在一起，而将 DOUT 端连接到相邻 MAX7219 的 DIN 端。在对 MAX7219 进行写操作时，需要使用 NO-OP 寄存器。例如将三个 MAX7219 组合使用，在对第三个 MAX7219 写入数据时，首先发送所需要的 16 位数据，其后跟两个空操作代码（×0××）。当 LOAD 引脚变为高电平时，数据将被锁存到所有器件中，而此时第三个芯片将接收到预期的数据，前两个芯片则只接受了空操作指令。

24.4 多个 LED 驱动实例

前面介绍了各种驱动多个 LED 数码管的方法，其中以外接 LED 驱动器最为方便，占用单片机资源少，而且程序控制简单。这里便以实例讲解如何使用 MAX7219 芯片和 51 系列单片机来驱动多个 LED 数码管。

24.4.1 LED 驱动器电路图

系统完整的电路图，如图 24-19 所示。这里的单片机选用 Atmel 公司的新型单片机 AT89S51，也可以采用其他兼容的 51 系列单片机，如 AT89S52、AT89C51、8051 等。该电路所需的元器件，如表 24-12 所示。

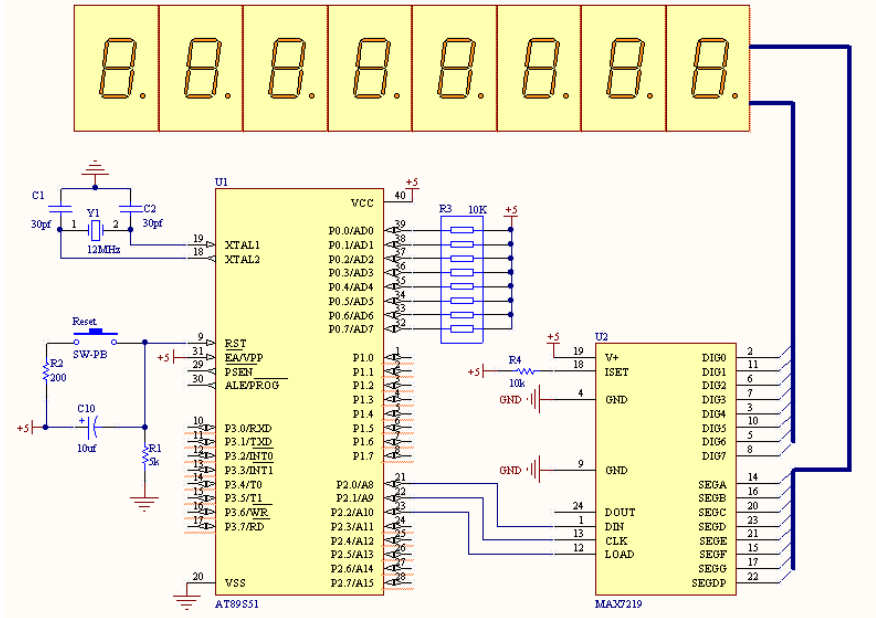


图 24-19 MAX7219 与 8051 单片机的接口

表 24-12 元器件列表

元 器 件	数 值	数 量
U1	AT89S51	1 个
U2	MAX7219	1 个
C1、C2	30pF	2 个
C10	10μF 电解	1 个
Y1	12.0000MHz 晶振	1 个
R1	5KΩ	1 个
R2	200Ω	1 个
R3	10KΩ上拉排阻	1 个
R4	10KΩ	1 个
LED	7 段共阴极 LED 数码管	8 个
SW-PB	复位开关	1 个

由于 MAX7219 采用的是三线串口方式，因此可以采用 8051 软件模拟的方式和 MAX7219 进行通信。在这里 AT89S51 的 P2.0 端口连接 MAX7219 的 DATA 端，P2.1 端口连接 CLK 端，P2.2 端口连接 LOAD 端。MAX7219 外接 8 个共阴极 LED 数码管。

24.4.2 程序设计

本例的程序功能是演示 MAX7219 的各个寄存器操作，以及控制 8 个 LED 数码管显示数字及字符。

1. 创建项目

这里采用 Keil C51 语言编写程序。具体操作步骤如下：

(1) 首先在 Keil μ Vision3 集成开发环境中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。

(2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S51，如图 24-20 所示。

(3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 24-21 所示。单击“是”按钮，完成工程的建立。

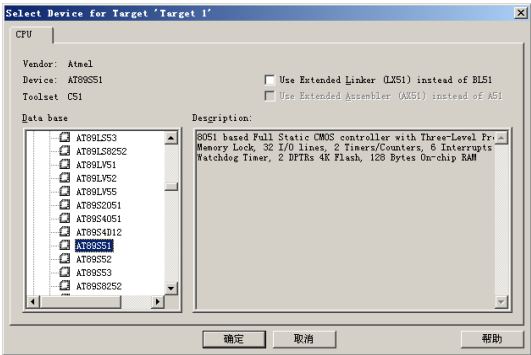


图 24-20 选择单片机 AT89S51

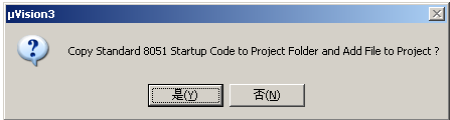


图 24-21 “ μ Vision3”对话框

(4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件。可以在其中输入程序代码。

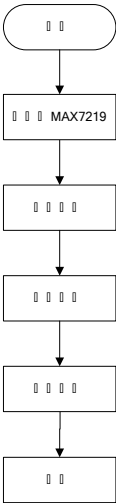


图 24-22 流程图

2. 程序代码

程序中主函数用于初始化 MAX7219，设置显示亮度，测试 LED 数码管以及数字和字符的显示。主程序的流程图，如图 24-22 所示。主程序代码示例如下：

```
#include <reg51.h> //头文件

void main(void)
{
    //初始化 MAX7219
    DATA=1;LOAD=1;CLK=1;
    WriteMAX7219(SCANLIMIT_REG, 7); //扫描 8 位数字
    WriteMAX7219(DECODE_REG, 0x00); //设置不译码方式
    StopShutdown(); //结束待机方式，正常工作
    StopLEDTest(); //结束显示测试，正常工作
    ClearLED(); //清零全部显示器
    SetLevel(MAX/2); //设置一半的亮度

    OpenLEDTest(); //显示测试
    StopLEDTest(); //结束显示测试

    LEDDisplay (0x01, '0'); //第 1 位 LED 显示 0
}
```

```

LEDDisplay (0x02, '1');           //第 2 位 LED 显示 1
LEDDisplay (0x03, '2');           //第 3 位 LED 显示 2
LEDDisplay (0x04, '3');           //第 4 位 LED 显示 3
LEDDisplay (0x05, '4');           //第 5 位 LED 显示 4
LEDDisplay (0x06, '5');           //第 6 位 LED 显示 5
LEDDisplay (0x07, '6');           //第 7 位 LED 显示 6
LEDDisplay (0x08, '7');           //第 8 位 LED 显示 7

ClearLED ();                       //关闭全部 LED 显示器

LEDDisplay (0x01, 'A');           //第 1 位 LED 显示 A
LEDDisplay (0x02, 'B');           //第 2 位 LED 显示 B
LEDDisplay (0x03, 'C');           //第 3 位 LED 显示 C
LEDDisplay (0x04, 'D');           //第 4 位 LED 显示 D
LEDDisplay (0x05, 'E');           //第 5 位 LED 显示 E
LEDDisplay (0x06, 'F');           //第 6 位 LED 显示 F
LEDDisplay (0x07, 'H');           //第 7 位 LED 显示 H
LEDDisplay (0x08, 'L');           //第 8 位 LED 显示 L

OpenShutdown ();                  //停机
StopShutdown ();                  //结束停机

while(1)                          //主循环
{
}

```

其中，分别调用了各个子函数来对 MAX7219 进行控制，实现对 8 个 LED 数码管的显示。下面分别介绍各预定义个子函数。

► 预定义

预定义部分位于 main 函数的前面，用于定义 MAX7219 的内部寄存器地址，以及端口所连接的 AT89S51 引脚。其中还对各个显示数字和字符对应的段码进行了定义，便于在程序中查表获得。预定义部分的代码示例如下：

```

#define DECODE_REG      0x09       //译码方式寄存器
#define INTENSITY_REG   0x0a       //亮度寄存器
#define SCANLIMIT_REG   0x0b       //扫描界限寄存器
#define SHUTDOWN_REG     0x0c       //停机寄存器
#define DISPLAYTEST_REG 0x0f       //显示测试寄存器

#define MIN      0x00             //最小显示亮度
#define MAX      0x0f             //最大显示亮度

sbit DATA=P2^0;                 //DATA 接口
sbit CLK=P2^1;                   //CLK 接口
sbit LOAD=P2^2;                  //LOAD 接口

static const struct
{
char  ascii;

```

```
char segs;
} Font[] = //定义并初始化显示数字和字符的段码数组
{
    {' ', 0x00},{'0', 0x7e},{'1', 0x30},{'2', 0x6d},{'3', 0x79},{'4', 0x33},
    {'5', 0x5b},{'6', 0x5f},{'7', 0x70},{'8', 0x7f},{'9', 0x7b},{'A', 0x77},
    {'B', 0x1f},{'C', 0x4e},{'D', 0x3d},{'E', 0x4f},{'F', 0x47},{'H', 0x37},
    {'L', 0x0e}, {'O', 0x7e},{'\0', 0x00}
};
```

➤ 字节发送函数

字节发送函数用于向 MAX7219 发送一个字节数据。该函数中采用 for 循环语句，按照串行结构，移位发送 8 位数据。字节发送函数的代码示例如下：

```
static void SendByte (unsigned char dataout)
{
    char i;
    for (i=8; i>0; i--) //循环发送
    {
        unsigned char mask = 1 << (i - 1);
        CLK=0; //CLK 变低电平
        if (dataout & mask) //发送一位数据
            DATA=1;
        else
            DATA=0;
        CLK=1; //CLK 变高电平
    }
}
```

➤ MAX7219 操作函数

MAX7219 操作函数用于向指定的 MAX7219 内部寄存器写入数据。该函数中，首先置 LOAD 为高电平，用于启动串行数据发送，接着依次发送 MAX7219 内部寄存器地址及数据，最后将数据锁存并结束操作。MAX7219 操作函数的代码示例如下：

```
static void WriteMAX7219 (unsigned char reg_number, unsigned char dataout)
{
    LOAD=1; //置 LOAD 为高电平，用于启动串行数据发送
    SendByte(reg_number); //发送内部寄存器地址
    SendByte(dataout); //发送数据
    LOAD=0; //置 LOAD 为低电平，锁存数据
    LOAD=1; //置 LOAD 为高电平，结束串行数据
}
```

➤ 停机方式函数

停机方式函数用于使 MAX7219 进入停机状态。该函数中使用 WriteMAX7219 函数写停机寄存器，使 MAX7219 进入停机模式。停机方式函数的代码示例如下：

```
void OpenShutdown (void)
{
    WriteMAX7219(SHUTDOWN_REG, 0); //写停机寄存器
}
```

➤ 结束停机方式函数

结束停机方式函数用于使 MAX7219 脱离停机状态，进行正常显示。该函数中使用 WriteMAX7219 函数写停机寄存器，使 MAX7219 脱离停机模式。结束停机方式函数的代码示例如下：

```
void StopShutdown (void)
{
    WriteMAX7219(SHUTDOWN_REG, 1); //写停机寄存器
}
```


➤ 显示测试函数

显示测试函数用于使 MAX7219 进入显示测试状态。该函数中使用 WriteMAX7219 函数写显示测试寄存器，使 MAX7219 进入显示测试模式。显示测试函数的代码示例如下：

```
void OpenLEDTTest (void)
{
    WriteMAX7219(DISPLAYTEST_REG, 1);           //写显示测试寄存器
}
```

➤ 结束显示测试函数

结束显示测试函数用于使 MAX7219 脱离显示测试状态，进行正常显示。函数中使用 WriteMAX7219 函数写显示测试寄存器，使 MAX7219 脱离显示测试模式。结束显示测试函数的代码示例如下：

```
void StopLEDTTest (void)
{
    WriteMAX7219(DISPLAYTEST_REG, 0);           //写显示测试寄存器
}
```

➤ 亮度设置函数

亮度设置函数用于设置 LED 数码管的显示亮度。该函数中使用 WriteMAX7219 函数写亮度寄存器，从而改变数码管的显示亮度。其中，亮度的取值范围为 0~15。亮度设置函数的代码示例如下：

```
void SetLevel (char ch)
{
    ch &= 0x0f;                                   //屏蔽参数中的多余位
    WriteMAX7219(INTENSITY_REG, ch);              //写亮度寄存器
}
```

➤ 关闭 LED 函数

关闭 LED 函数用于关闭所有的 LED 数码管。该函数中使用 WriteMAX7219 函数循环将 0x00 赋值给数字寄存器，使各个 LED 数码管熄灭。关闭 LED 函数的代码示例如下：

```
void ClearLED (void)
{
    char i;
    for (i=1; i <= 8; i++)                         //循环
        WriteMAX7219(i, 0x00);                     //写数字寄存器，关闭全部数码管
}
```

➤ 查找段码函数

查找段码函数用于查找并返回某个字符对应的 LED 显示段码。该函数中通过循环语句，在预定的段码表中进行查找。查找段码函数的代码示例如下：

```
static unsigned char FindCode (char character)
{
    char i;
    for (i = 0; Font[i].ascii; i++)                 //循环查表
        if (character == Font[i].ascii)
            return Font[i].segs;                   //返回对应的显示段码
    return 0;                                        //未找到，返回 0
}
```

➤ 显示字符函数

显示字符函数用于使 LED 数码管显示指定的数字或字符。该函数中使用 WriteMAX7219 函数写各个数字寄存器，其中用到了 FindCode 函数查找字符对应的段码。显示字符函数的代码示例如下：

```
void LEDDisplay (char digit, char character)
```

```
{
    WriteMAX7219(digit, FindCode(character));    //写数字寄存器
}
```

在程序设计完成后，便可以在 Keil μ Vision3 中进行编译仿真。当编译通过后，便可以将程序下载到 AT89S51 单片机中运行。从系统运行中可以看到 MAX7219 的显示测试操作、设置 LED 亮度、数字显示及字符显示等。

24.5 小结

本章详细介绍了 LED 数码管显示器件，包括 7 段共阳极 LED 数码管和 7 段共阴极 LED 数码管，然后介绍了 LED 的静态显示技术及其应用实例。本章还重点讲解了 LED 数码管的动态显示技术，包括静态驱动、动态驱动和 LED 驱动器驱动。最后通过一个具体的实例讲解了使用 LED 驱动器控制多个 LED 的显示。LED 数码管显示是单片机系统中常用的显示接口，读者应该熟练掌握其使用方法。

第 25 章 LCD 液晶显示模块

前面介绍的 LED 数码管只可以显示数字和某些字符，限制了其应用范围。在很多场合中需要显示多行字符、汉字或图形，于是液晶显示器便应运而生。液晶显示器（LCD）是一种功耗很低的显示器，其以优越的性能，越来越受到各方面的重视。液晶显示器的应用领域越来越多，在电子表、计算器、数码相机、计算机的显示器和液晶电视上都可以看到它的身影。

液晶显示模块是一种集成度比较高的显示组件，其英文名称为 LCD Module，可以简称为 LCM。液晶显示模块将液晶显示器件、控制器、PCB 电路板、背光源和外部连接端口等组装在一起，可以方便地用于需要液晶显示的场合。在现代的电子设计中，液晶显示模块的应用也越来越多。

本章将主要介绍液晶显示的结构和基本原理，液晶显示模块的类型及常用的液晶显示模块控制器；然后将通过一个实例讲解了使用液晶显示模式如何显示汉字和图形等。

25.1 LCD 液晶显示概述

LCD 液晶显示是依靠液晶显示器来进行数据或图形显示的。LCD 液晶显示弥补了 LED 显示效果不够美观、不能显示图形和汉字等缺点，因此液晶显示技术得到了越来越广泛的应用。下面介绍液晶显示方面的应用技术。

25.1.1 液晶的来源

液晶，顾名思义，就是固态之间的一种中间类状态。一般情况下，最常见的物质有三种形态，即固态、液态和气态。液晶是另外一种特殊的物质形态，由澳大利亚植物学者莱尼茨尔（Reinitzer）在 1888 年发现。此后，经过广泛的研究，液晶于 20 世纪 50 年代开始大规模使用。

液晶的成分是一种有机化合物，在一定的温度范围内，它既具有液体的流动性、黏度、形变等性质，又具有晶体的热（热效应）、光（光学各向异性）、电（电光效应）、磁（磁光效应）等物理性质。光线穿透液晶的路径由其分子排列所决定。人们通过研究发现，给液晶充电会改变它的分子排列，进而造成光线的扭曲或折射。液晶显示便是根据此原理来制成的。

按照分子结构排列的不同，液晶可以分为如下三类。

- 近晶相（Smectic）液晶：这种液晶的晶体颗粒为黏土状；
- 向列相（Nematic）液晶：这种液晶的晶体颗粒类似细火柴棒；
- 胆甾相（Cholestic）液晶：这种液晶的晶体颗粒类似胆固醇状。

这三种液晶的物理特性都不尽相同，目前，用于液晶显示器的是向列相（Nematic）液晶。

25.1.2 LCD 液晶显示器结构及原理

LCD 液晶显示器本身不发光，其通过调节光的亮度来达到显示效果，这是一种被动显示器。LCD 主要利用液晶的扭曲一向列效应制成，这是一种电场效应。下面首先介绍 LCD 液晶显示器的结构和原理，以及它的技术和工艺特点。这样才能在选购时有的放矢，更加科学合理地使用和维护。

液晶是一种有机复合物，由长棒状的分子构成。在自然状态下，这些棒状分子的长轴大致平行。LCD 大致有如下两个结构和功能上的特点。

1. 特点一

将液晶灌入两个列有细槽的平面之间，这两个平面上的槽互相垂直（90 度相交）。也就是说，如果一个平面上的分子南北向排列，则另一平面上的分子东西向排列，而位于两个平面之间的分子被强迫进入一种 90 度扭转的状态。由于光线顺着分子的排列方向传播，所以光线经过液晶时也被扭转 90 度。但当液晶上加一个电压时，分子便会重新垂直排列，使光线能直射出去，而不发生任何扭转。

2. 特点二

液晶显示依赖极化滤光片和光线本身。在实际生活中，自然光线是朝四面八方随机发散的。极化滤光片是一系列越来越细的平行线，这些线形成一张网，可以阻断不与这些线平行的所有光线，极化滤光片的线正好与第一个垂直，所以能完全阻断那些已经极化的光线。只有两个滤光片的线完全平行，或者光线本身已扭转与第二个极化滤光片相匹配，光线才得以穿透。

LCD 液晶显示器正是由这样两个相互垂直的极化滤光片构成，所以在正常情况下应该阻断所有试图穿透的光线。但是，由于两个滤光片之间充满了扭曲液晶，所以在光线穿出第一个滤光片后，会被液晶分子扭转 90 度，最后从第二个滤光片中穿出。另一方面，若为液晶加一个电压，分子又会重新排列并完全平行，使光线不再扭转，所以正好被第二个滤光片挡住。这样，便可以实现加电时将光线阻断，不加电则使光线射出，从而实现了 LCD 液晶的显示。

25.1.3 液晶显示模块的种类

液晶显示模块是以 LCD 液晶屏为核心，配合一定的控制电路，以达到方便使用显示组件的目的。根据 LCD 液晶屏可显示内容的不同，液晶显示模块可以分为如下三种。

1. 数显液晶模块

数显液晶模块中的显示部件是段型 LCD 液晶显示器件，其中为了使用的方便，还集成了专用的控制器和其他集成电路，其只能显示数字及一些标识符号，如图 25-1 所示。

2. 点阵字符液晶模块

点阵字符液晶模块的显示部件是点阵字符液晶显示器件，同样，集成有专用的行、列驱动器，控制器及必要的连接件、结构件等。这种液晶模块可以显示数字和西文字符，功能有所提高，但是不能显示图形，如图 25-2 所示。

3. 点阵图形液晶模块

点阵图形液晶模块的液晶显示器件是由连续的点阵像素构成的。因此不仅可以显示字符，而且可以显示连续、完整的图形，如图 25-3 所示。

这三种液晶显示模块中，功能最完善的是点阵图形液晶模块。其既能显示数字、字符，也能显示任意的汉字和图形等，而且还可以进行多行显示。其实，在不同的应用场合，每种液晶模块各有其用途，数显和字符液晶模块虽然功能少，但其成本低，适用于不需要图形显示的地方。



图 25-1 数显液晶模块



图 25-2 液晶点阵字符模块



图 25-3 点阵图形液晶模块

本章将以功能最强的点阵图形液晶模块为例来进行讲解。目前，市场上的点阵图形液晶模块，按照驱动方式的不同有如下三种类型可供选择。

- 行、列驱动型。模块只装配有通用的行、列驱动器，这种驱动器实际上只是对像素的一般驱动输出端。因而使用这种液晶模块的时候，还需要外接专用的控制器。
- 行、列驱动—控制型；这种液晶模块所用的列驱动器具有 I/O 总线数据接口，可以将模块直接连接到计算机的总线上，依靠计算机之间控制驱动，省去了专用控制器。
- 行、列控制型。这种液晶模块内置控制器。控制器是液晶驱动器与计算机或单片机的接口，它以最简单的方式受控于计算机或单片机。这种液晶模块具有自己的一套专用指令，并具有自己的字符发生器 CGROM。这种类型的液晶显示模块应用最为广泛。

本章介绍的液晶显示模块均是单色液晶，目前市场上还有一些彩色液晶模块和触摸屏液晶模块，其价格比较高、使用比较复杂，这里不做具体介绍。

25.1.4 液晶显示模块的优点

液晶显示最主要的优势是可以显示多行的汉字及图形。除了这一点外，使用液晶显示模块作为显示设备还具有其他很多优势，主要表现在如下几个方面。

- 体积小、重量轻。液晶显示模块通过显示屏上的电极控制液晶分子状态来达到显示目的，在重量上比相同显示面积的传统显示器件要轻得多。
- 功率消耗小。液晶显示模块的功耗主要消耗在其内部的电极和驱动芯片上。因而，对于相同的显示面积，液晶显示模块的耗电量比其他显示器件要小得多。
- 显示质量高。由于液晶显示模块每一个点在收到信号后就一直保持那种色彩和亮度，恒定发光，不像有些显示设备需要不断刷新亮点。因此，液晶显示模块画质高而不会闪烁，把眼睛疲劳降到了最低。
- 无电磁辐射。液晶显示模块的先天特点决定了其没有电磁辐射，这个优点使得液晶电视和计算机的液晶显示器都得到广泛推广。
- 简单方便的数字式接口。液晶显示模块都是数字式的，和单片机的接口十分简单，操作也十分方便。
- 应用范围广。液晶显示模块特别是点阵图形液晶模块，可以显示数字、字符、汉字和图形等，可适用于各种场合。

25.2 液晶显示模块控制接口

液晶显示模块（LCM）一般都内置 LCD 驱动器，其采用控制指令集来进行显示控制。这类 LCM 和单片机的接口比较简单，控制比较容易，因此得到了广泛的应用。一般来说，掌握一种液晶显示模块，便可以熟悉采用同类型驱动器的其他液晶显示模块的使用。下面重点介绍点阵图形液晶模块的使用，其可以显示数字、字符、汉字和图形等，功能比较全面。

25.2.1 LCD 控制驱动器 ST7920 概述

LCD 控制驱动器主要用于控制液晶的显示，市场上常见的有 ST7920、KS0066U、HD44780 等。这里介绍被广泛使用的 ST7920 液晶控制驱动器。

ST7920 是台湾矽创电子公司生产的中文图形控制芯片，它是一种功能极强的液晶控制模块，主要包括如下几方面的功能。

- 芯片内置 128×64—12 汉字图形点阵的液晶显示控制模块，用于显示汉字及图形。
- 芯片内置 8192 个中文汉字（16×16 点阵）。
- 128 个字符的 ASCII 字符库（8×16 点阵）。
- 64×256 点阵显示 RAM（GDRAM）。
- 芯片内部设计有 2MB 的中文字型 CGROM 和 64×256 点阵的 GDRAM 绘图区域，便

于简单、有效地显示汉字和图形。

- 芯片提供有 4 组可编程控制的 16×16 点阵造字空间。
- ST7920 由 32 个普通驱动器（common）及 64 个段驱动器 segment 组成。
- 芯片提供了 4 位并行、8 位并行、2 线串行及 3 线串行等多种接口方式，可以适应多种微处理器和单片机接口的需要。

使用 ST7920 可以方便地实现汉字、ASCII 码、点阵图形、自造字体的同屏显示，所有这些功能（包括显示 RAM、字符产生器以及液晶驱动电路和控制器）都包含在集成电路芯片里，因此，只要一个最基本的微处理器，例如 8051，就可以通过 ST7920 芯片来控制液晶显示。

25.2.2 ST7920 功能说明

ST7920 包含 ST7920-BIG5 和 ST7920-GB 两个型号，其中，ST7920-BIG5 内建 BIG-5 码繁体中文字型库，ST7920-GB 内建 GB 简体中文字型库。下面以内建简体中文字型库的 ST7920-GB 为例，介绍其功能及如何实现显示字符、汉字和图形等操作。

1. 基本操作功能

ST7920 提供了并行和串行两种控制方式，由外部 PSB 引脚来选择，具体方式如下：

- 当 PSB 输入高电平时，选择并行方式；
- 当 PSB 输入低电平时，选择串行方式。

ST7920 的读写操作主要用到两个 8 位的寄存器，一个是数据寄存器（DR），另一个是指令寄存器（IR）。其中，通过数据寄存器（DR）可以读取 DDRAM/CGRAM/GDRAM 及 IRAM 的值，通过指令寄存器（IR）可以实现不同的操作。对于待存取目标 RAM 的地址，也可以通过指令命令来选择。每次数据寄存器 DR 的操作应以上次选择的目标 RAM 为主体来进行读出或写入。通过 RS 和 RW 的状态可以选择不同的读写模式，具体方法如表 25-1 所示。

表 25-1 ST7920 寄存器读写状态

RS	RW	功能说明
L	L	MPU 写指令到指令寄存器（IR）
L	H	读出忙标志及地址计数器的状态
H	L	MPU 写指令到数据寄存器（DR）
H	H	MPU 从数据寄存器（DR）读出指令

2. 忙标志（BF）

忙标志（BF）用于指示 ST7920 指令是否执行完毕。一般来说，任何指令都需要一定的时间来处理，每个指令的处理时间不相同。ST7920 只有前面一条指令执行完毕后，才可以执行下一条指令。

当忙标志位 BF 为 1 时，表示内部操作正在进行，即处于忙状态而不接受新的指令。所以，每次输入新指令前，都要读取 BF 标志，只有当其为 0 时才可输入新指令进行执行。

3. 地址计数器（AC）

地址计数器（AC）用来存储 DDRAM/CGRAM/IRAM/GDRAM 之一的地址。其可以由设定指令寄存器（IR）来改变，之后只要读取或写入 DDRAM/CGRAM/IRAM/GDRAM 的值时，地址计数器（AC）的值就会自动加 1。

当 RS=0 和 RW=1 时，地址计数器（AC）的值会被读取到 DB6~DB0 中。

4. 中文字型 ROM（CGROM）及半宽字型 ROM（HCGROM）

ST7920 的中文字型 ROM 提供了 8192 个 16×16 点的中文字型图像及 128 个 16×8 点的数字符号图像，它使用两个字节来提供字型编码选择，把将要显示的字型码写入 DDRAM 中，硬件将自动依照编码从 CGROM 中提取将要显示的字型并将其显示在液晶屏幕上。

5. 自定义字型 RAM (CGRAM)

自定义字型 RAM (CGRAM) 用于设置自定义的字型或图像。ST7920 的字型产生 RAM 为用户提供了造字功能，可以提供 4 组 16×16 点的自定义图像空间，用户可以将内部没有提供的图像字符自行定义到 CGRAM 中，便可以 and CGRAM 中的定义一样通过 DDRAM 显示在液晶屏幕上。

6. 图标 RAM (IRAM)

ST7920 提供了 256 点的 ICON 图标显示，它分别由 16 组的 IRAM 地址来组成，每一个 IRAM 地址由 16 个位构成，每次写入一组 IRAM 时，需要先指定 IRAM 的地址，然后通过连续写入两个字节的资料来完成，先写入高字节 (D15~D8) 再写入低字节 (D7~D0)。

ICON RAM 的地址、数据及 segment 接脚对应关系，如图 25-4 所示。16×8 半宽字型符号表，如图 25-5 所示。

7. 显示数据 RAM (DDRAM)

显示数据 RAM 用于设置液晶显示的数据。其提供了 64×2 位的空间，最多可以控制 4 行 16 字(共 64 个字)的中文字型显示。当写入显示数据 RAM 时，可以分别显示 CGROM、HCGROM 与 CGRAM 的字型。

		ICON RAM 数据															
		高字节								低字节							
		D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
ICON RAM	0	SEG0	SEG1	SEG2	SEG3	SEG4	SEG5	SEG6	SEG7	SEG8	SEG9	SEG10	SEG11	SEG12	SEG13	SEG14	SEG15
	1	SEG16	SEG17	SEG18	SEG19	SEG20	SEG21	SEG22	SEG23	SEG24	SEG25	SEG26	SEG27	SEG28	SEG29	SEG30	SEG31
	2	SEG32	SEG33	SEG34	SEG35	SEG36	SEG37	SEG38	SEG39	SEG40	SEG41	SEG42	SEG43	SEG44	SEG45	SEG46	SEG47
	3	SEG48	SEG49	SEG50	SEG51	SEG52	SEG53	SEG54	SEG55	SEG56	SEG57	SEG58	SEG59	SEG60	SEG61	SEG62	SEG63
	4	SEG64	SEG65	SEG66	SEG67	SEG68	SEG69	SEG70	SEG71	SEG72	SEG73	SEG74	SEG75	SEG76	SEG77	SEG78	SEG79
地址	5	SEG80	SEG81	SEG82	SEG83	SEG84	SEG85	SEG86	SEG87	SEG88	SEG89	SEG90	SEG91	SEG92	SEG93	SEG94	SEG95
	6	SEG96	SEG97	SEG98	SEG99	SEG100	SEG101	SEG102	SEG103	SEG104	SEG105	SEG106	SEG107	SEG108	SEG109	SEG110	SEG111
	7	SEG112	SEG113	SEG114	SEG115	SEG116	SEG117	SEG118	SEG119	SEG120	SEG121	SEG122	SEG123	SEG124	SEG125	SEG126	SEG127
	8	SEG128	SEG129	SEG130	SEG131	SEG132	SEG133	SEG134	SEG135	SEG136	SEG137	SEG138	SEG139	SEG140	SEG141	SEG142	SEG143
	9	SEG144	SEG145	SEG146	SEG147	SEG148	SEG149	SEG150	SEG151	SEG152	SEG153	SEG154	SEG155	SEG156	SEG157	SEG158	SEG159
AC3 ~ AC0	A	SEG160	SEG161	SEG162	SEG163	SEG164	SEG165	SEG166	SEG167	SEG168	SEG169	SEG170	SEG171	SEG172	SEG173	SEG174	SEG175
	B	SEG176	SEG177	SEG178	SEG179	SEG180	SEG181	SEG182	SEG183	SEG184	SEG185	SEG186	SEG187	SEG188	SEG189	SEG190	SEG191
	C	SEG192	SEG193	SEG194	SEG195	SEG196	SEG197	SEG198	SEG199	SEG200	SEG201	SEG202	SEG203	SEG204	SEG205	SEG206	SEG207
	D	SEG208	SEG209	SEG210	SEG211	SEG212	SEG213	SEG214	SEG215	SEG216	SEG217	SEG218	SEG219	SEG220	SEG221	SEG222	SEG223
	E	SEG224	SEG225	SEG226	SEG227	SEG228	SEG229	SEG230	SEG231	SEG232	SEG233	SEG234	SEG235	SEG236	SEG237	SEG238	SEG239
	F	SEG240	SEG241	SEG242	SEG243	SEG244	SEG245	SEG246	SEG247	SEG248	SEG249	SEG250	SEG251	SEG252	SEG253	SEG254	SEG255

图 25-4 ICON RAM 的地址、数据及 segment 接脚对应关系

☒	☒	☒	♥	♠	♠	•	•	◉	◉	♂	♀	♫	♫	✳
▶	◀	↑	!!	¶	§	—	‡	↑	↓	→	←	⊥	++	▲
□	!	"	#	\$	%	&	'	()	*	+	,	-	.
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n
p	q	r	s	t	u	v	w	x	y	z	{	!	}	~

图 25-5 16×8 半宽字型符号表

ST7920 可以显示三种字型，分别是半宽的 HCGROM 字型，CGRAM 字型及中文 CGROM 字型。三种字型的选择由 DDRAM 中写入的编码来完成，在 0000H~0006H 的编码中将选择 CGRAM 的自定字型，02H~7FH 的编码中将选择半宽英文数字的字型，至于 A1 以上的编码将自动和下一个字节结合，组成两个字节的编码连成中文的编码 BIG5（A140~D75F）、GB（A1A0~F7FF），详细各种字型编码如下所述。

- 显示半宽字型：将 8 位的数据资料写入 DDRAM 中，范围为 02~7FH 的编码。
- 显示 CGRAM 字型：将 16 位的数据资料写入 DDRAM 中，总共有 0000H、0002H、0004H 和 0006H 4 种编码。
- 显示中文字型：将 16 位数据资料写入 DDRAM 中，范围为 A140H~D75FH 的编码（BIG5），范围为 A1A0H~F7FFH 的编码（GB）。

16 位数据写入 DDRAM 的操作，由连续写入两个字节来完成，即先写入高 8 字节（D15~D8），再写入低 8 字节（D7~D0）。

在 ST7920 中，DDRAM 数据（字元代码）、CGRAM 地址及 CGRAM 数据（显示图像）的对照关系，如图 25-6 所示。

8. 绘图显示 RAM（GDRAM）

绘图显示 RAM（GDRAM）用于液晶的图形显示。其提供 64×32 个字节的记忆空间（由扩充指令设定绘图 RAM 地址），最多可以控制 256×64 点的二维绘图缓冲空间。

在更改绘图 RAM 时，由扩充指令设定 GDRAM 地址，先设置垂直地址再设置水平地址（连续写入两个字节的的数据完成垂直与水平的坐标地址），再写入两个字节的的数据到绘图 RAM，而地址计数器（AC）会自动加 1，整个绘图 RAM 的写入步骤如下：

- （1）将垂直的坐标（Y）写入绘图 RAM 地址；
- （2）将水平的坐标（X）写入绘图 RAM 地址；
- （3）将高 8 位 D15~D8 写入到 RAM 中，即写入第一个字节；
- （4）将低 8 位 D7~D0 写入到 RAM 中，即写入第二个字节。

GDRAM 坐标地址与数据排列顺序对照表，如图 25-7 所示。

DDRAM数据				CGRAM地址				CGRAM数据 高字节								CGRAM数据 低字节														
B15~ B4				B3	B2	B1	B0	B5	B4	B3	B2	B1	B0	D5	D4	D3	D2	D1	D0	D7	D6	D5	D4	D3	D2	D1	D0			
0	X	00	X	00	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0		
					0	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	0	1	1	0	0	0	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0
					0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	X	01	X	01	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0		
					0	0	0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
					0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
					0	0	1	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	
					0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
					0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
					0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
					0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
					0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 25-6 DDRAM 数据（字元代码）、CGRAM 地址及 CGRAM 数据（显示图像）的对照关系

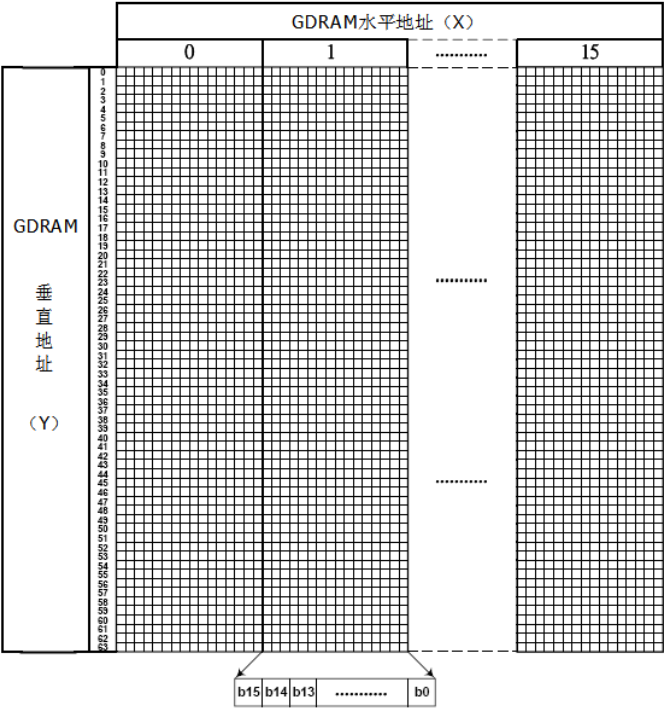


图 25-7 GDRAM 坐标地址与数据排列顺序对照表

9. LCD 驱动电路

LCD 驱动电路用于提供 33 个 common 及 64 个 segment 信号线来驱动 LCD 面板。其中，segment 数据从 CGRAM/CGROM 转换存储到 64 位的 segment 串列中，当 33 个 common 中的一个 common 输出时，相对应的 segment 数据将从 64 位的串列输出到 segment 驱动电路。

10. 游标/闪烁控制电路

ST7920 提供了硬件游标及闪烁控制电路，可以由地址计数器（Address Counter）的值来指定 DDRAM 中的游标位置或闪烁位置。

除了上述的功能外，ST7920 还提供了基本指令和扩充指令两套控制指令。下面分别进行介绍。

25.2.3 ST7920 基本指令集

ST7920 提供了 11 条基本指令，当 RE=0 时为基本指令集模式，下面分别介绍。

1. 清除显示指令

清除显示指令用于清除液晶屏的显示数据，其格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	0	0	0	1

清除显示指令执行的操作是将 DDRAM 填满 20H（space code），并且设定 DDRAM 的地址计数器（AC）到 00H，重设进入点设定，将 I/D 设为 1，即游标右移 AC 加 1。

2. 地址复位指令

地址复位指令可以将地址计数器复位，其格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	0	0	1	X

地址复位指令执行的操作是设定 DDRAM 的地址计数器（AC）到 00H，并且将游标移到开头原点位置，这个指令并不改变 DDRAM 的内容。

3. 进入设定点指令

进入设定点指令用于指定在数据的读取与写入时，设定游标的移动方向及指定显示的移位。指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	0	1	I/D	S

其中，I/D 是地址计数器递增递减的选择，当 I/D=1，游标右移，DDRAM 地址计数器（AC）加 1；当 I/D=0，游标左移，DDRAM 地址计数器（AC）减 1。

S 用于显示画面整体位移，当 S=1，I/D=1 时，画面整体左移；当 S=1，I/D=0 时，画面整体右移。

4. 显示状态开关指令

显示状态开关指令用于控制整体显示、游标、游标位置反白 ON/OFF 等操作。指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	1	D	C	B

其中，D 用于整体显示开/关控制，当 D=1，整体显示打开；当 D=0，整体显示关闭，但不改变 DDRAM 的内容。

C 用于游标开/关控制，当 C=1，游标显示打开；当 C=0，游标显示关闭。

B 用于游标位置反白开/关控制，当 B=1，游标位置反白打开，将游标所在之处的数据反白显示；当 B=0，游标位置反白关闭。

5. 游标或显示移位控制指令

游标或显示移位控制指令用于设定游标的移动与显示的移位控制，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	1	S/C	R/L	X	X

当 S/C=0，R/L=0 时，游标向左移动，地址计数器 AC=AC-1；
当 S/C=0，R/L=1 时，游标向右移动，地址计数器 AC=AC+1；
当 S/C=1，R/L=0 时，显示向左移动，且游标跟着移动，地址计数器 AC=AC；
当 S/C=1，R/L=1 时，显示向右移动，且游标跟着移动，地址计数器 AC=AC。
另外，游标或显示移位控制指令并不改变 DDRAM 的内容。

6. 功能设定指令

功能设定指令用于指令集的选择和控制方式的选择等，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	1	DL	X	RE	X	X

其中，DL 为 4/8BIT 控制方式选择，当 DL=1 时，为 8 位 MPU 控制方式；当 DL=0，为 4 位 MPU 控制方式。RE 为指令集选择控制，当 RE=1，将使用扩充指令集；当 RE=0，将使用基本指令集。
同一指令集不可同时改变 RE 及 DL，需先改变 DL，然后再改变 RE，才能确保 FLAG 正确设定。

7. 设定 CGRAM 地址指令

设定 CGRAM 地址指令用于将 CGRAM 地址送入地址计数器 AC 中，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

设定 CGRAM 地址指令在设定 CGRAM 地址到地址计数器（AC）时，AC 范围为 00H～3FH。需确认扩充指令中 SR=0（卷动地址或 RAM 地址选择）。

8. 设定 DDRAM 地址指令

设定 DDRAM 地址指令用于将 DDRAM 地址送入地址计数器 AC 中，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

设定 DDRAM 地址指令在设定 DDRAM 地址到地址计数器（AC）时，第一行 AC 范围为 80H～8FH，第二行 AC 范围为 90H～9FH，第三行 AC 范围为 A0H～AFH，第四行 AC 范围为 B0H～BFH。

9. 读取忙标志（BF）和地址指令

读取忙标志（BF）和地址指令用于读取忙标志（BF），可以用于确认内部动作是否完成，同时可以读出地址计数器（AC）的值。指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0

其中，当 BF=1，表示内部忙碌中，此时不能执行新指令，需等 BF=0 才可以执行新指令。

10. 写数据到 RAM 指令

写数据到 RAM 指令用于写入数据到内部的 RAM，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	1	0	D7	D6	D5	D4	D3	D2	D1	D0

当写入数据到内部的 RAM 时，会改变 AC 的值。每个 RAM 地址（CGRAM，DDRAM，IRAM……）都可以连续写入两个字节的数 据，当写入第二个字节时，地址计数器 AC 的值就会自动加 1。

11. 读取 RAM 的值指令

读取 RAM 的值指令用于读取 RAM 内部的数据，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	1	1	D7	D6	D5	D4	D3	D2	D1	D0

从内部的 RAM 读取数据时，当读取后会改变 AC 的值。当执行设定地址指令后（CGRAM，DDRAM，IRAM……），若要读取数据时需先 DUMMY READ 一次，才会读取到正确数据。第二次读取时则不需要 DUMMY READ，除非又执行了设定地址指令，才需要再次 DUMMY READ。

25.2.4 ST7920 扩充指令集

ST7920 扩充指令集提供了更高级的液晶显示控制。当 RE=1 时，为扩充指令集模式，ST7920 提供了 7 种扩充指令，下面分别介绍。

1. 待命模式指令

待命模式指令用于使 ST7920 进入待命模式，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	0	0	0	1

当 ST7920 进入待命模式后，执行其他任何指令都可以终止待命模式。待命模式指令并不改变 RAM 的内容。

2. 卷动地址或 RAM 地址选择指令

卷动地址或 RAM 地址选择指令用于 ST7920 卷动地址或 RAM 地址选择，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	0	0	1	SR

其中，当 SR=1，允许输入垂直卷动地址；当 SR=0，允许输入 IRAM 地址（扩充指令）及允许设定 CGRAM 地址（基本指令）。

3. 反白选择指令

反白选择指令用于选择 4 行中的任一行做反白显示，并可决定反白与否，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	0	1	R1	R0

其中，R1，R0 初值为 00，当第一次设定时为反白显示，再一次设定时为正常显示。R1 和 R0 的取值含义如下所示。

- 当 R1=0，R0=0，第一行反白或正常显示；
- 当 R1=0，R0=1，第二行反白或正常显示；

- 当 R1=1, R0=0, 第三行反白或正常显示;
- 当 R1=1, R0=1, 第四行反白或正常显示。

4. 睡眠模式指令

睡眠模式指令用于进入或脱离睡眠模式，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	0	0	1	SL	0	0

其中，当 SL=1，脱离睡眠模式；SL=0，进入睡眠模式。

5. 扩充功能设定指令

扩充功能设定指令用于指令集的选择和控制方式的选择等，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	0	1	DL	X	RE	G	X

其中，DL 为 4/8 BIT 控制选择，当 DL=1，为 8 位 MPU 控制方式；当 DL=0，为 4 位 MPU 控制方式。RE 为指令集选择控制，当 RE=1，为扩充指令集动作；当 RE=0，为基本指令集动作。G 为绘图显示控制，当 G=1，绘图显示 ON；当 G=0，绘图显示 OFF。

同一指令集不可同时改变 RE 及 DL、G，一般需要先改变 DL 或 G，然后再改变 RE，才能确保 FLAG 正确设定。

6. 设定 IRAM 地址或卷动地址指令

设定 IRAM 地址或卷动地址指令用于设定 ST7920 的 IRAM 地址或卷动地址，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

其中，SR=1，AC5~AC0 为垂直卷动地址；SR=0，AC3~AC0 为 ICON RAM 地址。

7. 设定绘图 RAM 地址指令

设定绘图 RAM 地址指令用于设定 ST7920 绘图 RAM 地址，指令的使用格式如下：

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
指令	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

设定 GDRAM 地址到地址计数器 AC 时，先设垂直地址再设水平地址（连续写入两个字节的数来完成垂直与水平的坐标地址）。垂直地址范围为 AC6~AC0，水平地址范围为 AC3~AC0。绘图 RAM 的地址计数器 AC 只会对水平地址（X 轴）自动加 1，当水平地址=0FH 时会重新设为 00H 但并不会对垂直地址做进位自动加 1，故连续写入多个数据时程序自动判断垂直地址是否需要重新设定。

25.2.5 ST7920 的操作方式

液晶控制驱动器 ST7920 提供了并行和串行两类指令操作方式。其中，并行操作方式又可以分为 8 位并行操作方式和 4 位并行操作方式两种。下面分别进行介绍。

1. 并行操作方式

当 PSB 引脚接高电平时，ST7920 则进入并行数据模式。在并行数据模式下，可由功能设定指令中的 DL 标志位来选择 8 位并行模式或 4 位并行模式，主控制系统将配合（RS, RW, E, DB0~DB7）来完成数据传送。

从一次完整的流程来看，当执行设定地址（CGRAM, DDRAM, IRAM 等）指令后，若要读取资料需先 DUMMY READ 一次，才能读取到正确资料，第二次读取时则不需 DUMMY

READ，除非又执行一次设定地址指令才需再次 DUMMY READ。

在 4 位并行模式中，每一个 8 位的指令或数据都被分成两个 4 位组来操作。高 4 个字节（DB7~DB4）的数据将会放在第一组的（DB7~DB4）部分，而低 4 个字节（DB3~DB0）的数据将会被放到第二组的（DB7~DB4）部分。在 4 位并行模式中，DB3~DB0 不使用。8 位并行模式和 4 位并行模式的时序图，分别如图 25-8 和图 25-9 所示。

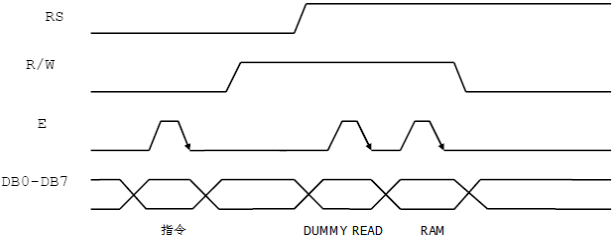


图 25-8 8 位并行模式的时序图

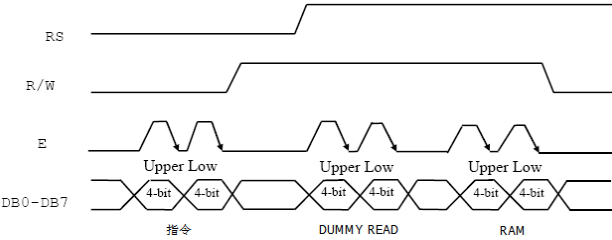


图 25-9 4 位并行模式的时序图

2. 串行操作方式

当 PSB 引脚接低电平时，ST7920 将进入串行数据模式。在串行模式下，将使用两根引脚时钟线（SCLK）和串行数据线（SID），来完成串行数据的传送。

当需要同时连接多个 ST7920 时，引脚 CS 用于使能选择。当 CS 接高电平时，此芯片才可以进行数据传输，如果 CS 接低电平，则 ST7920 的内部串行传输计数器与串行数据都将被重置，此时不能进行串行传输。如果系统中只有一个 ST7920，此时只需将 CS 接高电平即可。

在串行数据模式下，同样需要注意指令执行时间的问题，必须在前一个指令执行完毕后可以执行下一个指令。

从一个完整的串行数据传输过程来看，一开始需要先传输 5 个连续的 1，作为开始标志，此时传输计数器将被重置并且串行数据将被同步；然后再传送传输方向位 RW 和寄存器选择位 RS，最后第 8 位为 0。

在接收到同步信号及 RW 和 RS 的开始信号后，每一个 8 位的指令将被分为两个的字节来传输。较高 4 位（DB7~DB4）的指令数据将被放在第一个字节的低 4 位，而较低 4 位（DB3~DB0）的指令数据将被放在第二个字节的低 4 位。相隔的其余 4 位都为 0。串行数据传输的时序图，如图 25-10 所示。

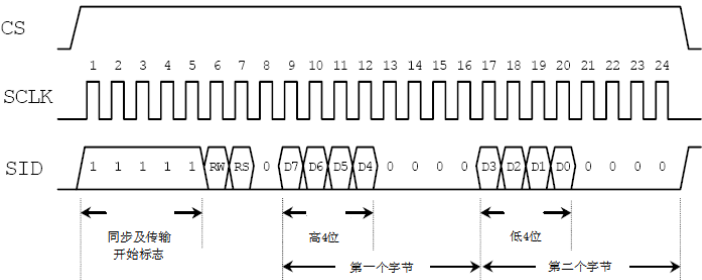


图 25-10 串行数据传输的时序图

25.2.6 图形点阵式液晶显示模块

市场上图形点阵式液晶显示模块有很多种，这里采用的液晶显示模块为北京宁和电子有限公司生产的 NH12864M 型 LCM 为例，如图 25-11 所示。

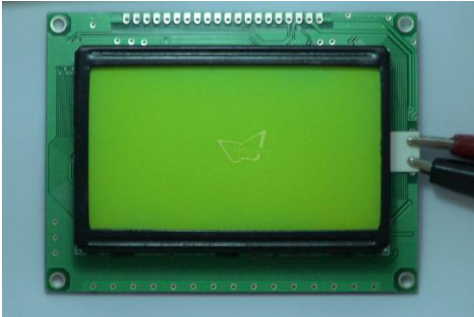


图 25-11 NH12864M 实物图

NH12864M 为 128×64 点的图形点阵式液晶模块，采用 LED 背光，其有 20 个引脚，分别定义如表 25-2 所示。

表 25-2 NH12864M 引脚说明

引 脚	符 号	电 平	功 能
1	VSS	0V	电源地
2	VDD	3.0~5.0V	正电源
3	V0	-	LCD 的操作电压
4	RS (CS)	H/L	H: 数据, L: 指令
5	R/W (SID)	H/L	H: 读, L: 写
6	E (CLK)	H/L	使能信号

续表

引 脚	符 号	电 平	功 能
7	DB0	H/L	数据总线
8	DB1	H/L	
9	DB2	H/L	
10	DB3	H/L	
11	DB4	H/L	
12	DB5	H/L	
13	DB6	H/L	
14	DB7	H/L	
15	PSB	H/L	H: 并行模式, L: 串行模式
16	NC	-	空
17	RST	L	复位信号, 低电平有效
18	NC	-	空
19	BLA	+5V	LED 背光电源
20	BLK	0V	

NH12864M 液晶显示模块采用的是 ST7920 控制芯片，其中各个引脚的功能及指令集和前面介绍的一致。因此，对这个液晶显示模块 NH12864M 的控制，只需要遵循前面的指令和操作方式即可。

25.2.7 图形点阵式液晶读写子函数

前面介绍了图形点阵式液晶控制器的指令集、操作方式及典型的液晶显示模块的引脚接口。下面按照 8 位并行接口操作方式来给出图形点阵式液晶显示模块的操作子函数，以方便读者调用。

这里假定 E 为液晶显示模块的读写使能信号，R_W 为液晶显示模块的读写选择信号，RS 为液晶显示模块的数据/指令信号，D_I 为液晶显示模块的数据指令选择信号。P1 端口作为数据指令输入端口。这些符号分别对应液晶显示模块的控制引脚，将在下面的子函数中使用。

1. 写指令子函数

写指令子函数用于向 ST7920 写各种操作指令。程序判断屏幕是否忙，如果忙则等待，如果不忙则执行写指令，并通过 E 来使能信号。写指令子函数程序示例如下：

```
void WCOM(unsigned char d)    //写指令程序
{
    R_W=1;
    D_I=0;                    //设置状态
Loop:
    P1=0xFF;
    E=1;                      //使能有效
    q=P1;                     //将 P1 口状态读入
    E=0;                      //关闭使能信号
    if (q&0x80)               //判断屏幕是否忙
    {
        goto Loop;           //若忙，循环等待
    }
    R_W=0;
    D_I=1;                    //写数据指令
    P1=d;                     //将数据送入 P1 口
    E=1;                      //使能信号开
    E=0;                      //使能信号关
}
```

2. 写数据子函数

写数据子函数用于向 ST7920 写各种操作数据。程序判断屏幕是否忙，如果忙则等待，如果不忙则执行写数据，并通过 E 来使能信号。写数据子函数程序示例如下：

```
void Dat(unsigned char d)
{
    R_W=1;
    D_I=0;                    //读状态指令
Loop:
    P1=0xFF;
    E=1;                      //使能
    q=P1;
    E=0;
    if (q&0x80)
    {
        goto Loop;           //判忙
    }
    D_I=1;                    //送数据指令
    R_W=0;
    P1=d;
    E=1;                      //使能信号开
```



```
E=0; //使能信号关
}
```

3. 初始化子函数

初始化子函数用于初始化 ST7920。该函数中使用了 WCOM 函数用于执行功能设定、开/关显示及游标右移等操作，其中还用到了 Clearlcd 清屏函数。初始化子函数程序示例如下：

```
void Imim()
{
    P0=0xff;
    P1=0xff;
    P2=0xff;
    P3=0xff;
    E=0;
    WCOM(0x38); //功能设定
    WCOM(0x08); //关显示指令
    WCOM(0x06); //游标右移
    Clearlcd(); //清屏
    WCOM(0x0c); //开显示指令
}
```

4. 基本指令集清屏子函数

基本指令集清屏子函数用于使用 ST7920 的基本指令集进行清屏操作。该函数中调用了 WCOM 函数来向 ST7920 写清屏指令。基本指令集清屏子函数程序示例如下：

```
void Clearlcd()
{
    D_I=0;
    R_W=0;
    P1=0x01;
    WCOM(0x01); //清除显示
}
```

5. 写汉字子函数

写汉字子函数用于在液晶屏上显示汉字。该函数调用了 WCOM 函数用于设定起始点，并使用 Dat 函数循环将汉字数组数据写入 ST7920。写汉字子函数程序示例如下：

```
void Hanz()
{
    WCOM(k); //设置起始点
    for (i=0; i<j; i++, p++)
    {
        wbyte=*p;
        Dat(wbyte); //将汉字数组数据送入
    }
}
```

6. 扩充指令集画图子函数

扩充指令集画图子函数用于使用 ST7920 的扩充指令集来进行绘图操作。该函数中，首先设定 ST7920 处于扩充指令集模式，然后通过页循环和列循环来送入图形数据。扩充指令集画图子函数程序示例如下：

```
void Tuxing()
{
    unsigned char data v;
    unsigned char data w;
    unsigned char data x;
    unsigned char data h;
```

```
WCOM(0x34); //功能设定
WCOM(0x36); //扩充功能设定
pp=p2;
pp=pp+16*y*2;
for (h=0;h<z/2;h++) //页循环
{
    v=Column|0x80;
    w=Page|0x80;
    WCOM(w); //送入起始页地址
    WCOM(v); //送入起始列地址
    Page++; //准备对下一页进行操作
    for (x=0;x<2*y;x++) //列循环
    {
        if (x<y)
        {
            wbyte=*p2;
            Dat(wbyte); //循环送入图形数据
            p2=p2+1;
        }
        else
        {
            wbyte=*pp;
            Dat(wbyte); //循环送入图形数据
            pp=pp+1;
        }
    }
}
```

7. 扩充指令集清屏子函数

扩充指令集清屏子函数用于使用 ST7920 的扩充指令集进行清屏操作。该函数中循环调用 Dat 函数按页和列将液晶屏清屏。扩充指令集清屏子函数程序示例如下：

```
void Clearlcd1()
{
    unsigned char data Disp_Page;
    unsigned char data i;
    unsigned char data j;
    for(i=0;i<64;i++) //共 32 页
    {
        Disp_Page=i|0x80; //设置页地址
        WCOM(Disp_Page); //送入页地址
        WCOM(0x80); //送入起始列地址
        for (j=0;j<32;j++)
        {
            Dat(0x00); //清屏
        }
    }
}
```

25.3 汉字及图形显示实例

液晶显示模块在现代的电子设计中的应用越来越广泛，特别是图形点阵式液晶显示模块，其不仅可以显示数字和字符，也可以用来显示中文和图形。

下面采用 NH12864M 来介绍如何使用液晶显示模块 LCM 来显示汉字、图形，以及在任意位置显示图形。程序中使用了前面介绍的图形点阵式液晶读写子函数。

25.3.1 电路设计

这里采用 Atmel 公司的 AT89S52 单片机作为控制器，同样也可以采用其他兼容的 51 系列单片机，例如 AT89S51、8051、AT89C51 等。整个电路的电路图，如图 25-12 所示。

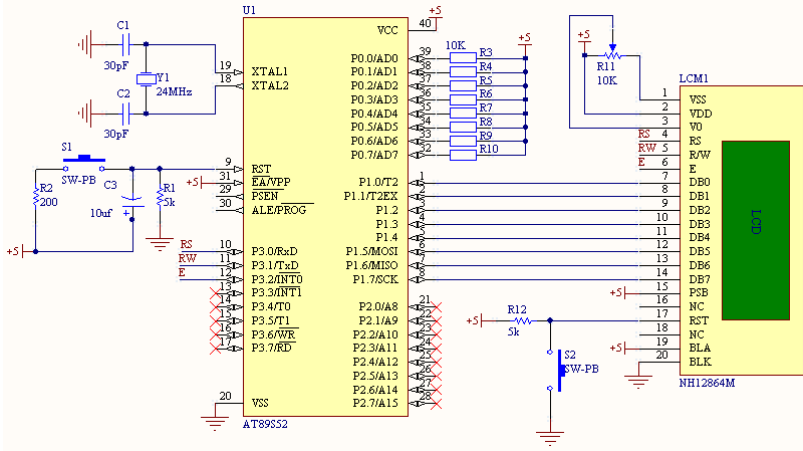


图 25-12 电路图

该电路所需的元器件，如表 25-3 所示。

表 25-3 元器件列表

元 器 件	数 值	数 量
U1	AT89S52	1 个
LCM1	NH12864M 液晶显示模块	1 个
C1、C2	30pF	2 个
C3	10μF 电解	1 个

续表

元 器 件	数 值	数 量
Y1	24.0000MHz 晶振	1 个
R1	5K Ω	1 个
R2	200 Ω	1 个
R3~R10	10K Ω	8 个
R11	10K Ω 电位器	1 个
S1、S2	复位开关	2 个

其中 R3~R10 为 8 个 10KΩ的上拉电阻，S1 用于单片机复位，S2 用于液晶显示模块 NH12864M 复位，可调电位器 R11 用于调节 LCD 显示屏的亮度。

PSB 接高电平，表示这里采用的是并行数据控制方式。另外，液晶显示模块的 E 接单片机的 P3.2 引脚，R_W 接单片机的 P3.1 引脚，D_I/RS 接单片机的 P3.0 引脚。

25.3.2 建立项目

这里采用 Keil C51 语言编写程序。具体操作步骤如下：

(1) 在 μ Vision3 中，选择“Project”→“New”→“μ Vision Project”命令，新建一个工程，并保存。

(2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 25-13 所示。

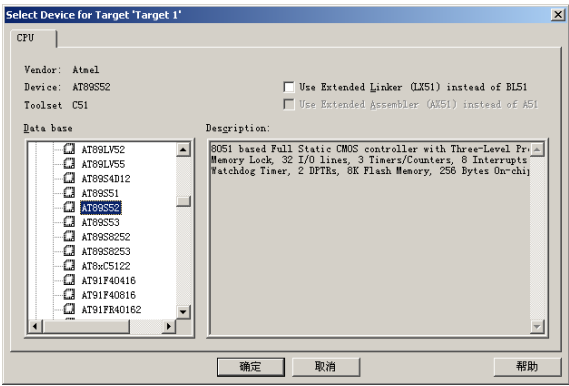


图 25-13 选择单片机 AT89S52

(3) 单击“确定”按钮，此时弹出“μ Vision3”对话框，如图 25-14 所示。单击“是”按钮，完成工程的建立。

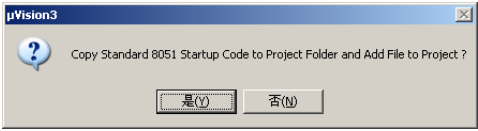


图 25-14 “μ Vision3”对话框

(4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件。可以在其中输入程序代码。

25.3.3 汉字显示实例

下面首先介绍如何使用液晶显示模块来显示多行汉字及字符。主程序的流程图，如图 25-15 所示。

使用液晶显示汉字的程序代码示例如下：

```
#include<reg52.h>           //头文件
#include<stdio.h>

unsigned data i;
unsigned char q;
unsigned char *p;           //建立数组指针
unsigned char wbyte;

sbit E=P3^2;                //读写使能信号，1 有效
sbit R_W=P3^1;              //读写选择信号；1 为读选通，0 为写选通
sbit RS=P3^0;
sbit D_I=P3^0;              //数据指令选择信号；1 为数据操作，0 为写指令或读状态

char code Hanzi[32]=       //汉字数组
{
    0x02,0x03,              //笑脸，心
    0xd6,0xd0,              //中
    0xbf,0xc6,              //科
    0xd4,0xba,              //院
    0xce,0xef,              //物
    0xc0,0xed,              //理
```

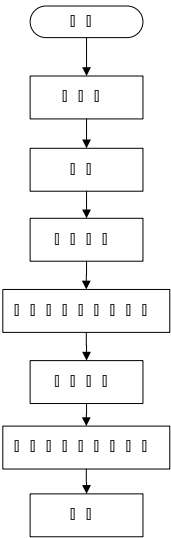


图 25-15 主程序流程图

```

0xcb,0xf9,          //所
0x03,0x02,          //心, 笑脸
0x54,0x45,          //T,R
0x4c,0x3a,          //L,:
0x30,0x31,          //0,1
0x30,0x2d,          //0,-
0x38,0x32,          //8,2
0x36,0x34,          //6,4
0x39,0x35,          //9,5
0x39,0x33,          //9,3
};
char code Hanzi1[28]= //汉字数组
{
    0xbb,0xb6,        //欢
    0xd3,0xad,        //迎
    0xc4,0xfa,        //您
    0xa3,0xa1,        //!
    0x20,0x20,        //空格
    0x20,0x20,
    0x46,0x41,        //F,A
    0x58,0x3a,        //X,:
    0x30,0x31,        //0,1
    0x30,0x2d,        //0,-
    0x38,0x32,        //8,2
    0x36,0x34,        //6,4
    0x39,0x34,        //9,4
    0x37,0x38,        //7,8
};

unsigned char data j;
unsigned char data k;

void main()
{
    Imim();           //初始化
    Clearlcd();

    p=Hanzi;           //指针指向数组的首地址
    j=32;              //所建立的第一个汉字数组中元素的个数
    k=0x80;            //起始点: 80h-8fh 为在第一行显示
    Hanz();            //在 1、3 行写汉字程序

    p=Hanzi1;
    j=28;              //所建立的第二个汉字数组中元素的个数
    k=0x92;            //起始点: 90h-9fh 为在第二行显示
    Hanz();            //在 2、4 行写汉字程序
    while(1)
    {
    }
}

```

在该程序中, 定义 E 接单片机 P3.2 引脚, R_W 接单片机 P3.1 引脚, D_I/RS 接单片机 P3.0 引脚。程序中还定义了 Hanzi 和 Hanzi1 为两个汉数组, 用于存放第一、三行和第二、四行显示的汉字和字符。其中的显示数据可以查看 ST7920 的字符集。

在主函数中, 首先初始化液晶屏, 然后进行清屏操作。接着将指针 p 分别指向 Hanzi 和 Hanzi1, 最后调用 Hanz 函数来在液晶屏上显示汉字和字符。

程序输入完毕后, 便可以编译, 如果程序没有错误, 便可以生成下载文件并下载到硬件中

执行。该程序执行时，液晶屏上显示的效果，如图 25-16 所示。

25.3.4 图形显示实例

下面介绍如何使用液晶显示模块来显示图形。主程序的流程图，如图 25-17 所示。



图 25-16 液晶汉字显示

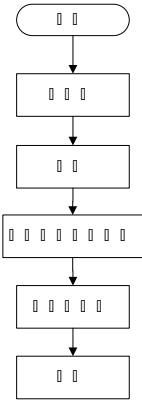


图 25-17 主程序流程图

使用液晶显示汉字的程序代码示例如下：

```
#include<reg52.h>
#include<stdio.h>

unsigned data i;
unsigned char q;
unsigned char *p2,*pp; //建立数组指针
unsigned char wbyte;
unsigned char Column; //定义列
unsigned char Page; //定义页

sbit E=P3^2; //读写使能信号，1 有效
sbit R_W=P3^1; //读写选择信号；1 为读选通，0 为写选通
sbit RS=P3^0;
sbit D_I=P3^0; //数据指令选择信号；1 为数据操作，0 为写指令或读状态

unsigned char data j;
unsigned char data k;

/*****扩充指令集画图*****/
unsigned char data z; //图形数组的页，为实际点阵行数
unsigned char data y; //图形数组的列，为实际点阵列数除以 8
unsigned char data Column; //起始点设定：为在液晶中显示的实际起始列
unsigned char data Page; //起始点设定：为在液晶中显示的实际起始行

char code Tu[64][16]= //图形数组，龙的标志
{
0xFF,0xFE,0xFE,0x80,0x00,0x00,0x18,0x00,0x00,0x00,0x00,0x00,0x01,0x7F,0x7F,0xFF,
0xE9,0x6E,0xEE,0x80,0x00,0x00,0x70,0x00,0x80,0x00,0x00,0x00,0x01,0x77,0x69,0x69,
0x96,0x96,0xEE,0x80,0x00,0x00,0x70,0x07,0x80,0x00,0x00,0x00,0x01,0x77,0x76,0x97,
0xE9,0x6E,0xE0,0x80,0x00,0x00,0xE6,0x1E,0x00,0x00,0x00,0x00,0x01,0x07,0x69,0x69,
0x96,0x96,0xFF,0x80,0x00,0x01,0xE8,0x78,0x00,0x00,0x00,0x00,0x01,0xFF,0x76,0x97,
0xE9,0x6E,0xFF,0x80,0x00,0x07,0xF7,0xFF,0xFF,0x80,0x00,0x00,0x01,0xFF,0x69,0x69,
0x96,0x96,0x00,0x80,0x00,0x0F,0xFF,0xFF,0xF8,0x00,0x00,0x00,0x01,0x00,0x76,0x97,
```

```

0xE9, 0x6F, 0xFE, 0x80, 0x00, 0x1F, 0xF7, 0xFF, 0xF0, 0x00, 0x00, 0x00, 0x01, 0x7F, 0xE9, 0x69,
0x96, 0x97, 0xFE, 0x80, 0x00, 0xFF, 0xFF, 0x0F, 0xF8, 0x00, 0x00, 0x00, 0x01, 0x7F, 0xF6, 0x97,
0xE9, 0x6E, 0x0E, 0x80, 0x01, 0xFF, 0xFE, 0xF6, 0xFF, 0x80, 0x00, 0x00, 0x01, 0x70, 0x69, 0x69,
0x96, 0x96, 0xEE, 0x80, 0x07, 0xFF, 0xF7, 0xFF, 0x1F, 0xF8, 0x00, 0x00, 0x01, 0x77, 0x76, 0x97,
0xE9, 0x6E, 0xEE, 0x80, 0xFF, 0xFF, 0xF0, 0x1E, 0xFF, 0xFE, 0x00, 0x00, 0x01, 0x77, 0x69, 0x69,
0x96, 0x96, 0xFE, 0x81, 0xFF, 0x1F, 0xF8, 0x01, 0xF7, 0xE1, 0x00, 0x00, 0x01, 0x7F, 0x76, 0x97,
0xE9, 0x6E, 0x00, 0x81, 0xFE, 0x1F, 0xFF, 0x00, 0x97, 0xF0, 0x78, 0x00, 0x01, 0x00, 0x69, 0x69,
0x96, 0x97, 0xFF, 0x80, 0x00, 0x78, 0x00, 0x00, 0x79, 0xF8, 0x68, 0x00, 0x01, 0xFF, 0xF6, 0x97,
0xE9, 0x6F, 0xFF, 0x80, 0x00, 0x70, 0x00, 0x00, 0x71, 0xFE, 0x7E, 0x00, 0x01, 0xFF, 0xE9, 0x69,
0x96, 0x96, 0x00, 0x80, 0x00, 0x70, 0x00, 0x00, 0x1F, 0xFE, 0x7F, 0x00, 0x01, 0x00, 0x76, 0x97,
0xE9, 0x6E, 0xFE, 0x80, 0x00, 0x70, 0x00, 0x00, 0x1E, 0xFE, 0x97, 0x80, 0x01, 0x7F, 0x69, 0x69,
0x96, 0x96, 0xEE, 0x80, 0x01, 0xF0, 0x00, 0x00, 0x0E, 0xF8, 0xFF, 0xE0, 0x01, 0x77, 0x69, 0x69,
0xE9, 0x6E, 0xEE, 0x80, 0x00, 0xF0, 0x00, 0x00, 0x1E, 0xFF, 0x9F, 0xF0, 0x01, 0x77, 0x69, 0x69,
0x96, 0x96, 0xE0, 0x80, 0x00, 0x60, 0x00, 0x00, 0x1E, 0xEF, 0xF9, 0xF0, 0x01, 0x07, 0x76, 0x97,
0xE9, 0x6E, 0xFF, 0x80, 0x00, 0x00, 0x00, 0x00, 0x16, 0xFF, 0x99, 0xF8, 0x01, 0xFF, 0x69, 0x69,
0x96, 0x96, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x79, 0xFF, 0x07, 0x01, 0x01, 0x77, 0x76, 0x97,
0xE9, 0x6E, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x7F, 0xF9, 0xFF, 0xFE, 0x01, 0x00, 0x69, 0x69,
0x96, 0x97, 0xFE, 0x80, 0x00, 0x00, 0x00, 0x01, 0xE7, 0xF9, 0x7F, 0xEF, 0x01, 0x7F, 0xF6, 0x97,
0xE9, 0x6F, 0xFE, 0x80, 0x00, 0x00, 0x00, 0x01, 0xEF, 0xF8, 0x07, 0x87, 0x01, 0x7F, 0xE9, 0x69,
0x96, 0x96, 0x00, 0x80, 0x00, 0x00, 0x00, 0x1F, 0x7F, 0x98, 0x07, 0x01, 0x01, 0x00, 0x76, 0x97,
0xE9, 0x6E, 0xEE, 0x80, 0x00, 0x00, 0x00, 0xFE, 0xFF, 0x90, 0x00, 0x00, 0x81, 0x77, 0x69, 0x69,
0x96, 0x96, 0xEE, 0x80, 0x00, 0x00, 0x01, 0xF7, 0xFF, 0x10, 0x00, 0x00, 0x01, 0x77, 0x76, 0x97,
0xE9, 0x6E, 0xFE, 0x80, 0x00, 0x00, 0x1E, 0xFF, 0xFE, 0x00, 0x80, 0x00, 0x01, 0x7F, 0x69, 0x69,
0x96, 0x96, 0x00, 0x80, 0x00, 0x00, 0x00, 0x1F, 0xEE, 0x1F, 0xFF, 0x08, 0x01, 0x00, 0x76, 0x97,
0xE9, 0x6F, 0xFF, 0x80, 0x00, 0x00, 0x8E, 0xFF, 0xE8, 0x1F, 0xFF, 0x00, 0x01, 0xFF, 0xE9, 0x69,
0x96, 0x97, 0xFF, 0x80, 0x00, 0x01, 0xF9, 0xF9, 0xE8, 0xFF, 0xEF, 0x00, 0x01, 0xFF, 0xF6, 0x97,
0xE9, 0x6E, 0x00, 0x80, 0x00, 0x06, 0xF7, 0xF9, 0x87, 0xF0, 0x01, 0xE0, 0x01, 0x00, 0x69, 0x69,
0x96, 0x96, 0xFE, 0x80, 0x00, 0x0F, 0x6F, 0xF9, 0x8F, 0xE0, 0x00, 0xE0, 0x01, 0x7F, 0x76, 0x97,
0xE9, 0x6E, 0xEE, 0x80, 0x00, 0x1F, 0x9F, 0x11, 0x0F, 0x80, 0x00, 0x70, 0x01, 0x77, 0x69, 0x69,
0x96, 0x96, 0xEE, 0x80, 0x00, 0x7F, 0xFF, 0x71, 0x0F, 0x00, 0x01, 0x91, 0x01, 0x77, 0x76, 0x97,
0xE9, 0x6E, 0xE0, 0x80, 0x00, 0x16, 0x7F, 0x10, 0x1F, 0x80, 0x00, 0xFE, 0x01, 0x07, 0x69, 0x69,
0x96, 0x96, 0x00, 0x80, 0x00, 0xFF, 0xFF, 0x60, 0x1F, 0xE0, 0x00, 0x07, 0xFF, 0x01, 0xFF, 0x76, 0x97,
0xE9, 0x6E, 0xFF, 0x80, 0x00, 0xFE, 0xFF, 0x00, 0x1F, 0xFF, 0x80, 0x1E, 0x01, 0xFF, 0x69, 0x69,
0x96, 0x96, 0x00, 0x80, 0x00, 0xFF, 0xEF, 0x00, 0x0F, 0xFF, 0xF8, 0x18, 0x01, 0x00, 0x76, 0x97,
0xE9, 0x6F, 0xFE, 0x80, 0x00, 0x96, 0xE7, 0x00, 0x07, 0xFF, 0xFE, 0x0E, 0x01, 0x7F, 0xE9, 0x69,
0x96, 0x97, 0xFF, 0x80, 0x01, 0xF9, 0xF1, 0x00, 0x00, 0xFF, 0x00, 0x01, 0x7F, 0xF6, 0x97,
0xE9, 0x6E, 0x0E, 0x80, 0x01, 0xF9, 0xF1, 0x00, 0x00, 0x1F, 0xFF, 0x00, 0x01, 0x70, 0x69, 0x69,
0x96, 0x96, 0xEE, 0x80, 0x01, 0xF9, 0xF8, 0x00, 0x00, 0x09, 0xFF, 0x80, 0x01, 0x77, 0x76, 0x97,
0xE9, 0x6E, 0xEE, 0x80, 0x01, 0xF9, 0xF8, 0x00, 0x00, 0x07, 0xFF, 0x80, 0x01, 0x77, 0x69, 0x69,
0x96, 0x96, 0xFE, 0x80, 0x01, 0x01, 0xF8, 0x00, 0x00, 0x01, 0xFF, 0xE0, 0x01, 0x7F, 0x76, 0x97,
0xE9, 0x6E, 0x00, 0x80, 0x01, 0xF9, 0x8E, 0x00, 0x00, 0x00, 0xFF, 0x80, 0x01, 0x00, 0x69, 0x69,
0x96, 0x97, 0xFF, 0x80, 0x01, 0xF9, 0xE6, 0x00, 0x00, 0x00, 0x7F, 0xE0, 0x01, 0xFF, 0xF6, 0x97,
0xE9, 0x6F, 0xFF, 0x80, 0x00, 0xFE, 0xF0, 0x00, 0x00, 0x06, 0x1F, 0xE0, 0x01, 0xFF, 0xE9, 0x69,
0x96, 0x96, 0x00, 0x80, 0x00, 0xFF, 0xF8, 0x00, 0x00, 0x07, 0xFF, 0xE0, 0x01, 0xFF, 0x76, 0x97,
0xE9, 0x6E, 0xFE, 0x80, 0x00, 0x96, 0xFF, 0x80, 0x00, 0x01, 0xFF, 0x80, 0x01, 0x7F, 0x69, 0x69,
0x96, 0x96, 0xEE, 0x80, 0x00, 0x7F, 0xF9, 0x60, 0x00, 0x01, 0xFF, 0x80, 0x01, 0x77, 0x76, 0x97,
0xE9, 0x6E, 0xEE, 0x80, 0x00, 0x7F, 0x78, 0x00, 0x00, 0x18, 0xFF, 0x80, 0x01, 0x77, 0x69, 0x69,
0x96, 0x96, 0xE0, 0x80, 0x00, 0xF9, 0x7E, 0x7F, 0x01, 0x10, 0xFF, 0x00, 0x01, 0x07, 0x76, 0x97,
0xE9, 0x6E, 0xFF, 0x80, 0x00, 0x1F, 0x9F, 0xFE, 0x1E, 0x70, 0xFF, 0x00, 0x01, 0xFF, 0x69, 0x69,
0x96, 0x96, 0xFF, 0x80, 0x00, 0x0F, 0xEF, 0xE0, 0x78, 0x79, 0xFF, 0x00, 0x01, 0xFF, 0x76, 0x97,
0xE9, 0x6E, 0x00, 0x80, 0x00, 0x07, 0x87, 0xE1, 0xF8, 0xFF, 0xFE, 0x00, 0x01, 0x00, 0x69, 0x69,
0x96, 0x97, 0xFE, 0x80, 0x00, 0x01, 0xF9, 0xFF, 0xF9, 0xFF, 0xF8, 0x00, 0x01, 0x7F, 0xF6, 0x97,
0xE9, 0x6F, 0xFE, 0x80, 0x00, 0x00, 0xF8, 0xFF, 0xFF, 0xFF, 0xF0, 0x00, 0x01, 0x7F, 0xE9, 0x69,
0x96, 0x96, 0x0E, 0x80, 0x00, 0x00, 0x6F, 0x7F, 0xFF, 0xFF, 0xE0, 0x00, 0x01, 0x70, 0x76, 0x97,
0xE9, 0x6E, 0xEE, 0x80, 0x00, 0x00, 0x1F, 0x87, 0xFF, 0xFF, 0x00, 0x00, 0x01, 0x77, 0x69, 0x69,
0x96, 0x96, 0xEE, 0x80, 0x00, 0x00, 0x0F, 0xF8, 0xFF, 0xF8, 0x00, 0x00, 0x01, 0x77, 0x76, 0x97,
0xFF, 0xFE, 0xFE, 0x80, 0x00, 0x00, 0x00, 0x0E, 0x70, 0x00, 0x00, 0x00, 0x01, 0x7F, 0x7F, 0xFF,
};

```

```

void main()
{
    Imim();                //初始化
    Clearlcd1();           //扩充指令集清屏

    p2=Tu;

```

```
z=64; //图形数组的页，为实际点阵行数
y=16; //图形数组的列，为实际点阵列数除以 8
Column=0; //起始点设定：为在液晶中显示的实际起始列
Page=0; //起始点设定：为在液晶中显示的实际起始行
Tuxing(); //龙的图形
while(1)
{
}
}
```

在该程序中，定义图形数组 **Tu**，其中存放了一个龙的图案对应的数据。一个图形对应的点阵数据可以通过相关的字模提取程序得到，网络上有各种该功能的程序，读者可以自己查找。

在主函数中，首先初始化液晶屏，然后使用扩充指令集进行清屏操作，接着将指针 **p2** 指向图形数组 **Tu**，然后调用 **Tuxing** 函数来在液晶屏上显示图形。

程序输入完毕后，便可以编译，如果程序没有错误，便可以生成下载文件并下载到硬件中执行。该程序执行时，液晶屏上显示的效果，如图 25-18 所示。

25.3.5 任意位置图形显示实例

下面介绍如何使用液晶显示模块在液晶屏的指定位置显示图形。主程序的流程图如图 25-19 所示。使用液晶在任意指定位置显示图形的程序代码示例如下：



图 25-18 图形显示实例

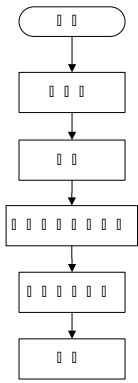


图 25-19 主程序流程图

```
#include<reg52.h>
#include<stdio.h>

unsigned data i;
unsigned char q;
unsigned char *p2,*pp; //建立数组指针
unsigned char wbyte;
unsigned char Column; //定义列
unsigned char Page; //定义页

sbit E=P3^2; //读写使能信号，1 有效
sbit R_W=P3^1; //读写选择信号；1 为读选通，0 为写选通
sbit RS=P3^0;
sbit D_I=P3^0; //数据指令选择信号；1 为数据操作，0 为写指令或读状态

unsigned char data j;
unsigned char data k;
```



```

/*****扩充指令集画图*****/
unsigned char data z;           //图形数组的页，为实际点阵行数
unsigned char data y;           //图形数组的列，为实际点阵列数除以 8
unsigned char data Column;      //起始点设定：为在液晶中显示的实际起始列
unsigned char data Page;        //起始点设定：为在液晶中显示的实际起始行

char code MyTu2[32][4]=        //图形数组，演示图形的任意位置显示
{
    0xAA,0x8A,0xE8,0xAA,0x00,0x5D,0xFC,0x00,0x2A,0xFB,0xBF,0xAA,0x01,0x51,0x15,0x40,
    0xAF,0xBF,0xFA,0xEA,0x05,0xDD,0x5D,0xD0,0x2E,0xF8,0x0F,0xBA,0x15,0x40,0x01,0x14,
    0xBF,0x8E,0x00,0xFE,0x75,0x05,0x00,0x54,0x3A,0x03,0x00,0x6E,0x74,0x00,0x00,0x17,
    0xFE,0x02,0xBE,0x3F,0x44,0x01,0xC4,0x15,0xEC,0x0F,0x82,0x3B,0x74,0x01,0x07,0x17,
    0xFC,0x3B,0x8E,0x3F,0x54,0x41,0x5C,0x15,0xEC,0x63,0xA8,0x3B,0x54,0x11,0x00,0x17,
    0xEE,0x0A,0x00,0x2F,0x54,0x01,0x00,0x57,0x3B,0x03,0x02,0x7E,0x11,0x01,0x00,0x54,
    0xBB,0xC3,0x81,0xFE,0x1D,0xC0,0x05,0xDC,0x2E,0xFA,0x2F,0xBA,0x07,0x77,0x75,0x70,
    0xAB,0xEB,0xFB,0xEA,0x01,0xD7,0x55,0xC0,0x2A,0xBF,0xFF,0xAA,0x00,0x17,0x74,0x00,
};

void main()
{
    Imim();                     //初始化
    Clearlcd1();                //扩充指令集清屏

    z=32;                       //图形数组的页，为实际点阵行数
    y=4;                        //图形数组的列，为实际点阵列数除以 8
    Column=5;                   //起始点设定：为在液晶中显示的实际起始列，一列占一个汉字
    Page=0;                     //起始点设定：为在液晶中显示的实际起始行
    p2=MyTu2;
    OneTuxing();                //显示一个小的图标
    while(1)
    {
    }
}

```

在该程序中，定义图形数组 `MyTu2`，其中存放了一个一个小的图标对应的图形数据。在主函数中，首先初始化液晶屏，然后使用扩充指令集进行清屏操作。接着指定图形显示的位置并将指针 `p2` 指向图形数组 `MyTu2`，然后调用 `OneTuxing` 函数来在液晶屏上指定位置显示图形。

这里用到了 `OneTuxing` 函数，这是一个自定义的函数，作用和 `Tuxing` 函数差不多。只不过 `OneTuxing` 函数更适合于在任意指定的位置显示一个小的图形。在该函数中，首先设定 `ST7920` 处于扩充指令集模式，然后通过页循环和列循环来送入图形数据。

```

void OneTuxing()
{
    unsigned char data v;
    unsigned char data w;
    unsigned char data x;
    unsigned char data h;

    WCOM(0x34);                //功能设定
    WCOM(0x36);                //扩充功能设定

    for(h=0;h<z;h++)           //页循环
    {
        v=Column|0x80;
        w=Page|0x80;

        WCOM(w);               //送入起始页地址
        WCOM(v);               //送入起始列地址
    }
}

```

```
Page++; //准备对下一页进行操作
for (x=0;x<y;x++) //列循环
{
    wbyte=*p2;
    Dat (wbyte); //循环送入图形数据
    p2++;
}
}
```

程序输入完毕后，便可以编译，如果程序没有错误，便可以生成下载文件并下载到硬件中执行。该程序执行时，液晶屏上显示的效果，如图 25-20 所示。



图 25-20 任意位置图形显示效果

25.4 小结

本章详细介绍了液晶和液晶显示模块的结构及原理，并对常用的 LCD 驱动控制器的指令和功能进行了介绍，其中给出了用于图形点阵式液晶读写的详细子函数。最后利用一款采用这个驱动器的液晶显示模块，来介绍如何控制其显示汉字和图形等。近年来，液晶显示模块应用越来越广泛。熟练掌握本章内容对读者以后的设计很有帮助。

第 26 章 D/A 转换实例

在实际的控制系统中，经常需要控制一些模拟信号，例如精确可调的电压电流输出、显示器亮度的调节及激光二极管偏置电压等。而一般的 8051 单片机外部总线接口为数字信号，无法直接产生需要的模拟信号。因此，需要将单片机的控制信号转换为期望的电压或电流等模拟信号。这便用到 D/A 转换器。D/A 转换器提供了良好的数字接口，可以和单片机的并行 I/O 接口直接相连，由单片机来控制，使其输出要求的模拟量电压或模拟量电流等。本章主要介绍了 D/A 转换的原理，D/A 转换器的类型及技术参数。本章还介绍了一款性能优秀的电压型 D/A 转换器。最后通过一个实际应用的例子，D/A 转换器的控制原理。

26.1 D/A 转换概述

D/A (Digital/Analogue) 转换也称为数/模转换，其进行的是数字量和模拟量之间的转换。一般来说，能够提供数字量转换为模拟量的器件，称为 D/A 转换器或数/模转换器。使用 D/A 转换技术可以利用成熟方便的数字处理技术，来产生和精确控制各种模拟量。

随着半导体工艺的发展，各种类型的 D/A 转换器层出不穷。这些 D/A 转换芯片在精度、转换速度、可靠性和方便性等方面都日趋成熟，并且其特有的数字接口可以很方便地和单片机相连，便于控制，从而很好地满足了各种测控系统的需求。下面首先介绍 D/A 转换及 D/A 转换器的相关知识。

26.1.1 D/A 转换原理

D/A 转换的基本功能是将一个数字量按照比例转换成模拟量（电压或电流）。D/A 转换所采用的基本方法是将数字量转化成二进制数据，其每一位产生一个相应的电压或电流，而这个电压或电流的大小正比于相应的二进制位的权，最后将这些电压或电流相加并输出。

由于一个数字量是由数字代码按位组合而成的，每一位数字代表一定的“权”，一个数字与对应的权相结合，就代表了一个具体的数值，把所有的数值相加，便得到该数的数字量。D/A 转换器正是使用了这一点，其要求该数字量对应一个模拟量，则只需将各位数字量分别转换成相应的模拟量，然后将所有的模拟量相加，所得到的总和即是该数字量相对应的模拟量。

下面首先介绍一下 D/A 转换器的原理。以 8 位的电压型 D/A 转换器为例，其结构原理图如图 26-1 所示。其主要由加权电阻解码网、数字量控制的电子开关组合和由运算放大器构成的电流电压转换器三部分组成。

在图 26-1 中，电子开关组和输入二进制数 $D_0 \sim D_7$ 相对应，当某一位为 1 时，相应的电子开关闭合，基准电压 V_{ref} 连接加权电阻解码网络，使某一个支路电阻上有电流通过；当某一位为 0 时，相应的电子开关断开，该支路电阻上无电流通过。

在加权电阻解码网络中，各个分支的电阻值与输入的二进制数据 $D_0 \sim D_7$ 的权相对应，权大的电阻值小，权小的电阻值大。最后根据各个权位的情况，通过求和及电流电压转换电路，得到总和的模拟电压输出值。

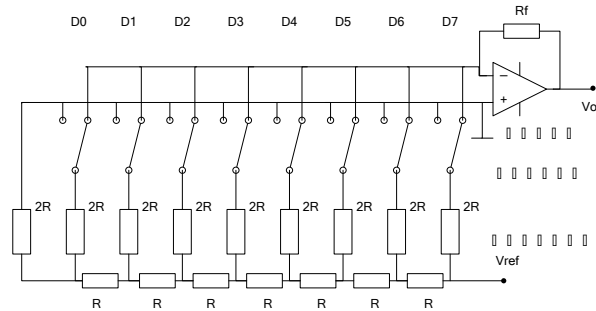


图 26-1 D/A 转换器原理图

26.1.2 D/A 转换器的类型

按照不同的用途及特性，D/A 转换器有不同的类型。下面分别介绍。

1. 按照输出模拟量的类型区分

按照输出模拟量的类型，可以分为电压型 D/A 转换器和电流型 D/A 转换器两种。分别介绍如下：

➤ 电压型 D/A 转换器，常用的有 AD558、MAX521、TLV5620 等。

电压型 D/A 转换器在芯片内部一般不直接从电阻阵列中输出电压，经常采用内置的输出放大器来实现低阻抗的输出。这种电压输出型 D/A 转换器没有放大部分的延迟，可以做到很高的速度，且一般用于高阻抗负载的情况下。

➤ 电流型 D/A 转换器，常用的有 DAC0832、THS5641、AD7523 等。

这种 D/A 转换器虽然输出的是电流的形式，但也可以调整为电压输出。在输出端加一个运算放大器，通过运算放大器构成电流—电压的转换电路，从而实现将转换器输出的电流转变为电压输出。也可以在输出端接一个负载电阻，通过压降来实现电流—电压转换，不过这种方法用的比较少。

2. 按照数字量的位数区分

按照 D/A 转换器的数字量输入端的位数，可以分为 8 位、10 位、12 位、16 位等 D/A 转换器。数字量的位数也表示了 D/A 转换器的分辨率，即输入数字量发生单位数字的变化的时候，对应的模拟量输出的大小。

- 常用的 8 位 D/A 转换器有 DAC0832、AD558 等；
- 常用的 10 位的 D/A 转换器有 AD7533、AD7397 等；
- 常用的 12 位的 D/A 转换器有 AD7564、AD7394、TLC5638 等；
- 常用的 16 位的 D/A 转换器有 AD420、AD421、MAX5621 等。

3. 按照接口类型区分

按照 D/A 转换器接口类型的不同，可以分为并行 D/A 转换器和串行 D/A 转换器两种。常用的并行 D/A 转换器有 AD7523、DAC0832、AD7226 等，串行 D/A 转换器有 AD7564、AD7243、MAX521 等。

一般来说，并行 D/A 转换器的数字—模拟转换速度要比串行 D/A 转换器要高。因此，对于速度要求比较高的场合，应该优先选用并行 D/A 转换器。这里以并行 D/A 转换器为例进行介绍。

并行 D/A 转换器的结构原理如图 26-2 所示。其内部将数字量的二进制位和电子开关相对应，输入数字量的各位数字决定了电子开关的状态，该位为 1 则开关闭合，该位为 0 则开关断开。标准电源提供的电流通过闭合的电子开关流入解码网络，解码网络则将标准电源电压转换成相应的电流。最后，通过求和放大器输出和输入数值大小成比例的模拟电压。

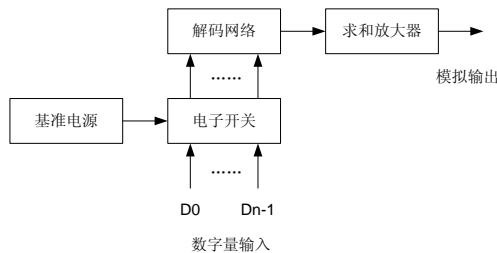


图 26-2 并行 D/A 转换器结构

在并行 D/A 转换器中，根据解码网络结构的不同，可以分为权电阻译码网络、T 型网络及变形权电阻译码网络等种类。目前市场上比较多的 D/A 转换芯片均采用 R-2R T 型解码网络和 TTL 或 MOS 型电流开关结构。

并行 D/A 转换器转换速度快的原因是在数字量的输入端，各个数字位是同时输入到转换器相应的端口的。整个器件的转换时间取决于电源电压的建立时间和各个权位的求和时间，而这段时间一般都很短，一般可以达到纳秒或微秒量级。

4. 按照内部是否集成锁存器区分

按照内部是否集成锁存器来区分，D/A 转换器可以分为内部没有锁存器的 D/A 转换器和芯片内部集成有锁存器的 D/A 转换器。

由于数字—模拟量的转换是需要一定的时间来完成的，在这段转换时间内，要求 D/A 转换器输入端的数字量输入保持不变，否则会导致错误的输出。因此，在 D/A 转换器的数字量输入端口前面应该设置锁存器，将数字量的输入数据保护起来。市场上的 D/A 转换器有的内置锁存器，而有的则没有锁存器。

内部没有锁存器的 D/A 转换器，其结构简单，但必须将额外的锁存器加到芯片数字量的输入端口之前，人工实现锁存。这类芯片有 AD7520、DAC800 等。

芯片内部集成有锁存器的 D/A 转换器，使用起来比较方便，不需要增加额外的器件。这类 D/A 转换器有 AD7542、AD7549、DAC8031 等。

26.1.3 D/A 转换器的技术参数

D/A 转换器的技术参数很多，如分辨率、量程、转换建立时间等。下面分别介绍这些技术参数的具体含义。

1. 分辨率

分辨率是 D/A 转换器对输入数字量变化的敏感程度，当输入数字量发生单位数字变化的时候，即 LSB 位产生一次变化时，所对应的输出模拟量（电压或电流）的变化量。分辨率与输入数字量的位数有关。

如果 D/A 转换器输入数字量的位数为 n，则 D/A 转换器的分辨率为 2^{-n} ，例如 8 位 D/A 转换器的分辨率为 1/266，10 位 D/A 转换器的分辨率为 1/1024，……。数字量的位数越多，分辨率就越高。

2. 精度

在理想情况下，D/A 转换器的精度与分辨率有关，即相当于分辨率的大小。

其实 D/A 转换器的转换精度是个比较复杂的问题，不仅与 D/A 转换芯片的内部结构有关，还与接口电路的配置有关。当外电路中的接口器件或电源有比较大的误差的时候，会造成比较大的 D/A 转换误差，D/A 转换的精度也就相应的降低了。

3. 标称满量程（NFS）

标称满量程（NFS）是指 D/A 转换器中，相应于数字量的标称值 2^n 的模拟输出量。

4. 实际满量程（AFS）

实际满量程（AFS）是指实际输出的模拟量。D/A 转换器的实际数字量为 2^n-1 ，要比标称值小一个 LSB，即实际满量程（AFS）要比标称满量程（NFS）小一个 LSB 的增量。

5. 转换建立时间

转换建立时间是描述 D/A 转换器运行快慢的一个参数，其值为数字量输入到模拟量输出至终值误差 $\pm (1/2)$ LSB（最低有效位）时所需要的时间。转换建立时间表明了 D/A 转换器的数字—模拟转换速度。

电流输出型的 D/A 转换器的转换建立时间比较短，而对于电压输出型 D/A 转换器，由于内部的运算放大器需要有一定的延迟时间，转换建立时间要长一些。

6. 尖峰

尖峰指的是 D/A 转换器的数字量输入端的数字信号发生变化的时刻，在输出端产生的瞬间误差。

以上这些技术参数反映了 D/A 转换器的主要性能，是选购 D/A 转换器时主要参考的标准。另外还有其他一些技术参数，这里不再一一介绍。

26.2 高速 D/A 转换芯片 AD558

市场上的 D/A 转换器有很多种型号，在选择时需要考虑其的精度、量程范围以及转换建立时间等参数，同时还要注意使用的方便性。下面介绍一款由美国 ANALOG DEVICES 公司推出的，具有高的转换速度及简单方便的控制接口的电压输出型 D/A 转换芯片 AD558。

26.2.1 AD558 简介

AD558 的引脚分配图，如图 26-3 所示。AD558 的内部功能框图，如图 26-4 所示。AD558 的主要性能指标如下：

- 8 位并行数字量输入宽度；
- 两种电压的输出范围，分别为 $0\sim+10V$ 和 $0\sim+2.56V$ ；
- 相对精度 $\pm (1/2)$ LSB；
- 高速 $1\mu s$ 输出转换建立时间；
- 单一电源供电，电源电压的范围 $+4.5V\sim+16.5V$ ；
- 内部具有基准电压源，不用外接基准源；
- 内部集成有数据输入锁存器。
- 低功耗，75mW。

AD558 提供了非常简单的接口，下面分别介绍各个引脚的主要功能。

- +VCC (Pin11)：电源电压输入端，电压范围为 $+4.5V\sim+16.5V$ ；
- GND (Pin12~Pin13)：电源地线；
- DB0~DB7 (Pin1~Pin8)：8 位并行 TTL 数字量的输入端；
- \overline{CE} (Pin9)：使能信号输入端；
- \overline{CS} (Pin10)：片选信号输入端；
- Vout Select (Pin14)：输出电压选择端；
- Vout Sense (Pin15)：输出电压敏感端；
- Vout (Pin16)：模拟电压输出端，可以输出 $0\sim+10V$ 或 $0\sim+2.56V$ 的模拟电压。

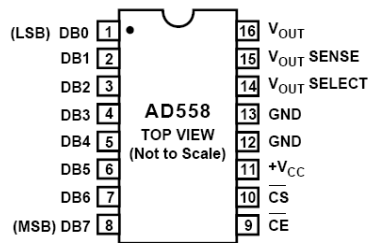


图 26-3 DIP 封装的 AD558

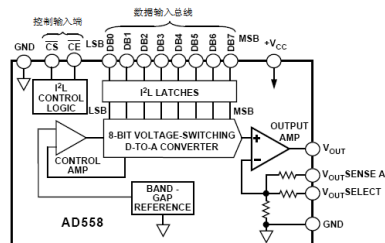


图 26-4 AD558 的功能框图

26.2.2 AD558 电压输出模式

AD558 提供了 0~+10V 和 0~+2.56V 两种模拟电压输出范围，供不同的场合使用。其输出范围的选择依赖于简单的外部接线方式，使用起来很方便。AD558 无须外部调整便可以获得 $\pm 1/2\text{LSB}$ 精度的输出电压。下面分别介绍如何 AD558 的两种电压输出模式。

1. 0V~+2.56V 模拟电压输出

利用 AD558 可以获得 0V~+2.56V 的输出电压，其外部接线图，如图 26-5 所示。此时芯片的电源供电电压范围很宽，为+4.5V~+16.5V。AD558 无须再连接任何外部器件，无须做任何调整，便可以 and 8 位数据总线连接获得需要的模拟输出电压。

在 0V~+2.56V 模拟电压输出模式下，AD558 的输入编码与模拟电压输出的关系，如表 26-1 所示。

表 26-1 AD558 的输入编码与电压输出

数字量输入编码			0V~+2.56V 电压输出
二进制码	十六进制码	十进制码	
0000 0000	00	0	0V
0000 0001	01	1	0.010V
0000 0010	02	2	0.020V
0000 1111	0F	15	0.150V
0001 0000	10	16	0.160V
0111 1111	7F	127	1.270V
1000 0000	80	128	1.280V
1100 0000	C0	192	1.920V
1111 1111	FF	255	2.550V

2. 0V~+10V 模拟电压输出

利用 AD558 可以获得 0V~+10V 的输出电压，其外部接线图如图 26-6 所示。此时芯片的电源供电电压范围为+11.4V~+16.5V。同样，AD558 无须再连接任何外部器件，无须做任何外部调整，便可以 and 8 位数据总线连接获得需要的模拟输出电压。

在 0V~+10V 模拟电压输出模式下，AD558 的输入编码与模拟电压输出的关系，如表 26-2 所示。

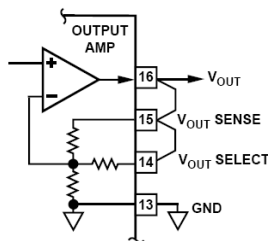


图 26-5 AD558 获得 0V~+2.56V 电压输出

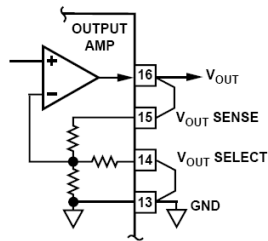


图 26-6 AD558 获得 0V~+10V 电压输出

表 26-2 AD558 的输入编码与电压输出

数字量输入编码			0V~+10V 电压输出
二进制码	十六进制码	十进制码	
0000 0000	00	0	0V
0000 0001	01	1	0.039V
0000 0010	02	2	0.078V
0000 1111	0F	15	0.586V
0001 0000	10	16	0.625V
0111 1111	7F	127	4.961V
1000 0000	80	128	5.000V
1100 0000	C0	192	7.500V
1111 1111	FF	255	9.961V

26.2.3 AD558 的数据锁存

AD558 是一款内部集成数据输入锁存器的 D/A 转换器，在数字—模拟量转换时可以将输入数据锁存，减少干扰，而且可以方便地将多个 AD558 连接到 8 位数据总线上。

AD558 的数据锁存由片选信号 \overline{CS} 和芯片使能信号 \overline{CE} 来控制。AD558 的控制逻辑功能图，如图 26-7 所示，其控制逻辑真值表如表 26-3 所示。

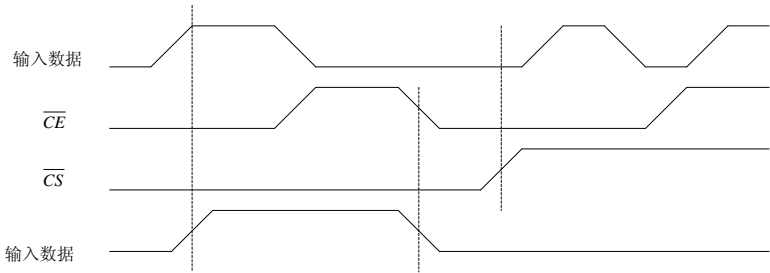


图 26-7 AD558 的控制逻辑功能图

表 26-3 AD558 的控制逻辑真值表

输入数据	\overline{CE}	\overline{CS}	DAC 数据	状 态
0	0	0	0	透明
1	0	0	1	透明
0	g	0	0	开始锁存
1	g	0	1	开始锁存
0	0	g	0	开始锁存

续表

输入数据	\overline{CE}	\overline{CS}	DAC 数据	状 态
1	0	g	1	开始锁存
X	1	X	数据保持	锁存
X	X	1	数据保持	锁存

注：X=任意值；g=TTL 信号的上升沿。

从表 26-3 中可以看出，只有当 \overline{CS} 和 \overline{CE} 全部为低电平时，AD558 才可以接收数据总线上的数据，实现从数字量到模拟量的转换。而当 \overline{CS} 和 \overline{CE} 中有一个为高电平时，锁存器工作，将输入端的数据锁存到内部存储器，此时输出电压不再随数据总线上的数据而变化。

数模转换芯片 AD558 的使用十分简单方便，覆盖了常用的电压输出范围，而且精度及可靠性也很高，转换速度也很快。更为重要的是，AD558 无须外接复杂的基准电压源，无须调试，直接便可以获得所需的模拟输出电压，能够适用于一般的控制系统的要求，性能价格比很高。

使用 AD558 可以节约很多的电路设计调试时间,降低电路的复杂性,从而加快设计周期并减轻电子设计工程师的工作量。这是一款很好的 D/A 转换芯片,下面的实例中也将采用这个芯片。

26.3 光通信电压调制电路实例——电路部分

在实际的测控系统中,很多设备或器件需要输入模拟的电压或电流来进行控制。在光通信领域中,经常需要对信号的强度及相位进行精确的控制。其中使用到光纤型相位调制器,可以通过改变输入的电压,使通过的光信号的相位得到改变。

下面将介绍如何使用 D/A 转换芯片来设计光纤型相位调制器的驱动电路,实现精确的调制电压输出。

26.3.1 相位调制的原理

相位调制在光通信领域中主要用于改变输出光的相位。目前比较多的是利用线性电光效应实现光的相位调制,其利用模拟电压在调制器中产生的电场来改变器件的折射率,从而达到改变光相位的目的。

1. 线性电光效应简介

有些双折射晶体如 ADP (磷酸二氘钾), LiNbO_3 (铌酸锂) 等,在外加电场 E 之后,晶体的折射率将发生线性变化,这种现象称为线性电光效应。

理论分析表明,在某方向上加电场之后, $n_o \rightarrow n'_o = n_o + \Delta n_o$; $n_e \rightarrow n'_e = n_e + \Delta n_e$, 而且

$$n'_o = n_o + \frac{1}{2} n_o^3 \gamma E \quad (1)$$

$$n'_e = n_e - \frac{1}{2} n_e^3 \gamma E \quad (2)$$

由以上两式可知

$$n'_o - n'_e = (n_o - n_e) + (\Delta n_o - \Delta n_e) = (n_o - n_e) + n_o^3 \gamma E \quad (3)$$

式 (2) 中第一项为自然双折射,由晶体决定;第二项中 γ 称为线性电光系数,由晶体及外电场方向决定, E 为外电场强度。因为第二项与外电场强度成正比,故称为线性电光效应。由 (3) 式可得到相位差

$$\delta = \frac{2\pi}{\lambda} \cdot l (n'_o - n'_e) = \frac{2\pi}{\lambda} \cdot l \cdot [(n_o - n_e) + n_o^3 \gamma E] \quad (4)$$

式 (3) 中自然双折射项产生固定的位相差,在使用中往往想办法消去,于是 $\delta \propto E$, 即相位差将正比于外电场变化。假定外电压是横向电场,且晶体厚度为 d , 则有

$$E = \frac{V}{d} (V/m) \quad (5)$$

$$\delta = \frac{2\pi}{\lambda} \cdot l \cdot n_o^3 \gamma \cdot \frac{V}{d} = \frac{2\pi}{\lambda} \cdot n_o^3 \gamma \cdot V \quad (6)$$

这里认为晶体的长度 $l = d$ 。

当 $\delta = \pi/2$ 时,要求外电压为 $V_{\pi/2}$, 从 (6) 式有

$$V_{\pi/2} = \frac{\lambda}{4} \cdot \frac{1}{n_o^3 \gamma} \quad (7)$$

此时, $V_{\pi/2}$ 电压加到双折射晶体之上,双折射晶体就起到一个 $\lambda/4$ 波片的作用,于是 $V_{\pi/2}$ 电压又称为 $V_{\pi/4}$ 电压。

当 $\delta = \pi$ 时,此时的外加电压定义为半波电压: V_π , 从 (6) 式可以求出

$$V_{\pi} = \frac{\lambda}{2} \bullet \frac{1}{n_o^3 \gamma}$$

(8)

显然，双折射晶体这时起 $\lambda/2$ 波片作用， V_{π} 又称为 $V_{\lambda/2}$ ，即半波电压。

2. 光纤型相位调制器

由前面的介绍可知，相位调制器的半波电压正比于晶体的厚度。对于一般的“体调制器”，具有较大的尺寸，这样在给整个晶体施加外电场时，需要相当高的电压，才能使通过的光波受到调制。高电压的产生和控制比较困难，因此给使用带来了麻烦。光纤型相位调制器的出现解决了这个难题。光纤型相位调制器的结构示意图，如图 26-8 所示。

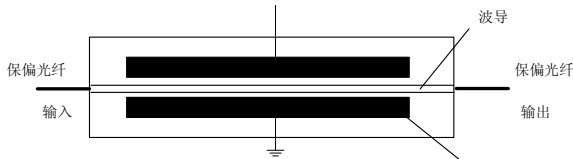


图 26-8 光纤型相位调制器

从图 26-8 中可知，光纤型波导相位调制器主要采用由介质构成的平面波导，利用光波导器件把光波限制在微米量级的波导区中，并使其沿一定的方向传播。这样，便可以使用线性电光效应改变介质的折射率，从而实现对光相位的控制。这种电光波导型相位调制器的主要优点是体积小，调制需要的半波电压小，便于电路控制。

26.3.2 电压调制系统

下面介绍一种常用的电压调制系统，系统的原理图，如图 26-9 所示。系统采用重复频率为 1MHz 的脉冲激光器，即每隔 1 μ s 产生一个光脉冲。光脉冲在经过光纤相位调制器时被调制，并将调制后的光脉冲信号输出。

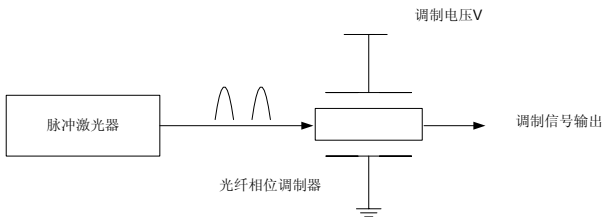


图 26-9 系统原理图

根据前面相位调制原理的介绍，加载在光纤相位调制器两端的电压和光脉冲相位的改变之间，具有如下的关系。

- 如果不加任何电压，则通过的光脉冲的相位不变；
- 如果加半波电压的一半，则通过的光脉冲的相位将被改变 $\pi/2$ ；
- 如果加半波电压，则通过的光脉冲的相位将被改变 π ；
- 如果加半波电压的 3/2，则通过的光脉冲的相位将被改变 $3\pi/2$ 。

电压调制系统主要根据外部的控制信号输入，对光脉冲进行不同顺序的调制。系统调制状态的转移，如图 26-10 所示。系统上电的时候，不加载调制电压，即不进行调制；随后根据外部控制信号，而分别进入不同的调制状态；当控制信号消失的时候，系统重新返回到零调制状态。

在这个系统中有 4 种调制状态，分别对应 4 种调制模式，分别介绍如下：

- 调制模式 1，采用 1/2 半波电压 $V_{\pi/2}$ 进行调制；
- 调制模式 2，采用半波电压 V_{π} 进行调制；
- 调制模式 3，采用 3/2 半波电压 $V_{3\pi/2}$ 进行调制；

➤ 调制模式 4，采用阶梯波进行调制，即循环用 0 、 $V_{\pi/2}$ 、 V_{π} 和 $V_{3\pi/2}$ 进行调制。

由于系统中采用的是脉冲激光器，为了实现对每个脉冲进行不同的调制，这里需要产生的调制信号也是脉冲形式的，如图 26-11 所示。

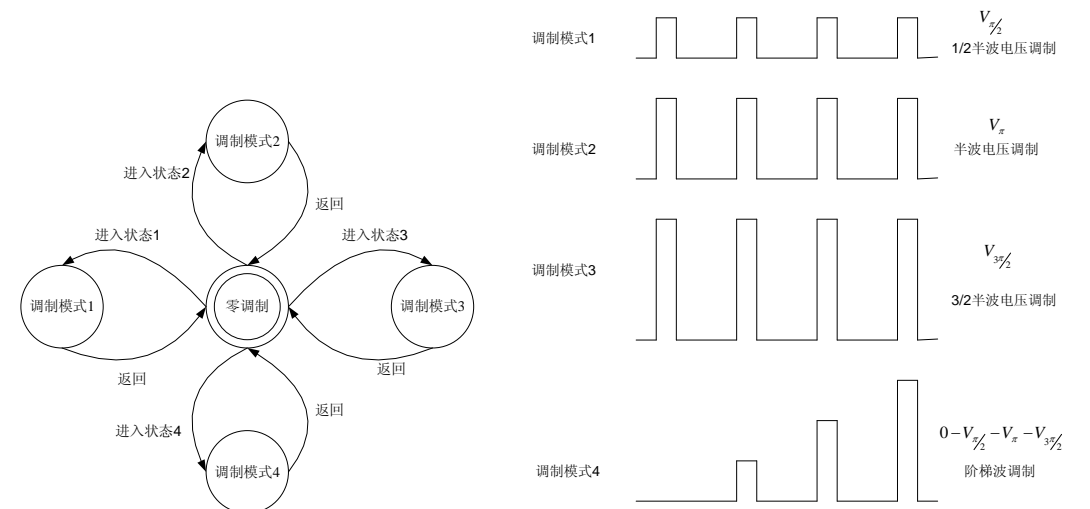


图 26-10 调制状态的转移

图 26-11 调制模式

26.3.3 电路图

电压调制系统的电路图，如图 26-12 所示。该电路图中所使用的元器件的参数及数量，如表 26-4 所示。

表 26-4 元器件列表

元 器 件	数 值	数 量
U1	AT89S52	1 个
U2	ADG201HS	1 个
U3、U4、U5	AD558	3 个
C1、C2	30pf	2 个
C3	10μf 电解	1 个

续表

元 器 件	数 值	数 量
C11~C16	10μf	6 个
S1、Mode1~Mode4	按键开关	5 个
R1	5kΩ	1 个
R2	200Ω	1 个
R3	10kΩ 排阻	1 个
R4~R7	1kΩ	4 个
P1	BNC 接头	1 个
Y1	24MHz	1 个

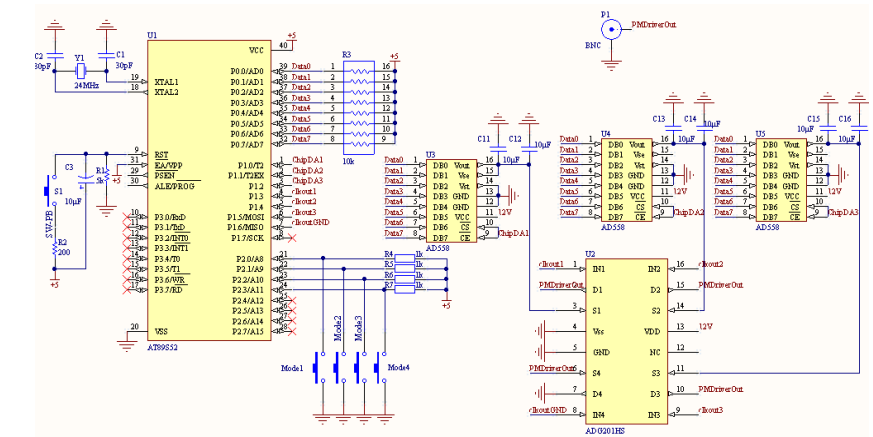


图 26-12 电路图

该电路中使用 Atmel 公司的 AT89S52 作为控制单元，4 个按键开关 Model1~Mode3 分别对应于 4 种调试模式的输入。电路中采用 AD558 作为调制电压生成器，以及 ADG201HS 作为模拟开关。下面对电路的每一部分进行详细的介绍。

1. 模拟电压产生电路

在这里使用的光纤相位调制器的半波电压为 4.2V，因此，需要进行调制的电压有 0V、2.1V、4.2V 和 6.3V。为了对系统的各个参数进行精确控制，这里不采用固定的电压，而采用 D/A 转换器输出可调的电压。

系统中采用 Analog Devices 公司的电压输出型数模转换芯片 AD558，选用电压输出范围为 0~+10V 的工作方式。参照前面介绍的连线图，其中只要将 Vout 和 Vout SENSE 连接在一起，将 Vout SELECT 和 GND 连接在一起，再加上电源及数字量输入便可以了。

AD558 的数字量输入端由 AT89S52 的 P0 口作为数据总线，通过内部的锁存器来控制对每个 D/A 转换器进行参数设置。

2. 模拟开关电路

电压调制系统中分别采用三个 AD558 芯片来产生三个调制电压，分别为 2.1V、4.2V 和 6.3V。系统中需要根据调制模式，选择调制电压输出加载到光纤相位调制器上，这便需要进行调制电压的切换。输出电压的切换可以采用两种方法，机械开关和电子模拟开关。

机械开关主要依靠机械触点的闭合与断开来实现输出电压的选通。每个电压输出后面都通过一个开关，如继电器，来接到光纤相位调制器的电极上。将需要输出电压的后面的继电器闭合，而其余继电器断开，则只选通其中一个电压输出。这种方法受限于继电器的开关速度，不能进行快速切换，并且开关时候的噪声也比较大，这里不适合采用。

电子模拟开关是依靠晶体管或场效应管等的导通、截止来实现开关的。电子模拟开关的主要特点是开关速度快、体积小、功耗低。这里为了达到很快的开关切换速度，采用的便是电子模拟开关。

下面介绍美国 Analog Devices 公司推出的一款高速、四单刀单掷模拟开关器件 ADG201HS。ADG201HS 的引脚封装和逻辑功能，分别如图 26-13 和图 26-14 所示。

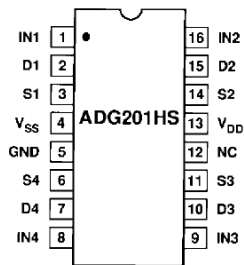


图 26-13 ADG201HS 的引脚封装图

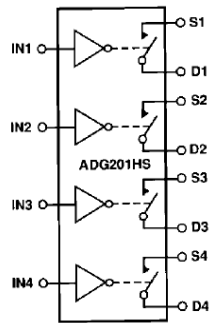


图 26-14 ADG201HS 的逻辑功能

ADG201HS 由 4 组单刀单掷模拟开关构成,可以分别控制 4 路的导通与断开。当控制端 IN1 为低电平的时候, S1 和 D1 导通, 当控制端 IN1 为高电平的时候, S1 和 D1 断开。其余三组具有相同的结构。该芯片的主要性能参数如下:

- 高速导通和关闭, 50ns;
- 低导通电阻, 30 Ω ;
- 单电压供电, $V_{DD}=+10.8V\sim+16.5V$;
- 宽的电压操作范围, 在 S 和 D 端口的模拟电压范围为 $0\sim V_{DD}$ 。

这里需要注意的是, 在 D/A 转换芯片 AD558 的输出端和地线之间需要并接两个 10μf 的电容。这是因为 ADG201HS 进行高速开关和断开的时候, 由于寄生电容充放电的影响, 会在模拟开关的输入端口累积大量的电荷, 这样在切断调制电压输出的时候, 电压并不是立即消失, 而是经过很长的放电时间才降为 0, 这样便影响了开关的切换速度。这两个并接的 10μf 电容可以降低这种影响, 从而提高模拟开关的切换速度。

3. 系统控制

整个系统的状态控制由单片机 AT89S52 来完成, 其通过并行口对模拟开关和 D/A 转换器进行控制。这里使用了 P0、P1、P2 共三个并行口, 功能分别介绍如下:

P0 端口用做数据总线。本例中的三个 D/A 转换器 AD558 的数字量的输入端是并接在一起的, 共用一个数据总线, 由 P0 口提供。这样可以减少并行端口的数目。

P1 端口用于控制 D/A 转换器的选通和调制输出。P1 端口每个引脚的功能定义, 如表 26-5 所示。

表 26-5 P1 端口设置

端 口	名 称	用 途
P1.0	ChipDA1	为 0 时选通第一个 D/A 转换器, 为 1 时锁定第一个 D/A 转换器
P1.1	ChipDA2	为 0 时选通第二个 D/A 转换器, 为 1 时锁定第二个 D/A 转换器
P1.2	ChipDA3	为 0 时选通第三个 D/A 转换器, 为 1 时锁定第三个 D/A 转换器
P1.3	clkout1	为 0 时选通调制电压 1 输出, 为 1 时切断调制电压 1

续表

端 口	名 称	用 途
P1.4	clkout2	为 0 时选通调制电压 2 输出, 为 1 时切断调制电压 2
P1.5	clkout3	为 0 时选通调制电压 3 输出, 为 1 时切断调制电压 3
P1.6	clkoutGND	为 0 时选通零电压输出, 为 1 时切断零电压

P2 端口用于调制模式输入。P2.0~P2.3 分别对应调制模式 1~调试模式 4。当按键按下的时候, 选用某个调试模式, 如果多个按键按下或没有按键按下, 则不输出任何调制电压。

26.4 光通信电压调制电路实例——程序部分

本例程序所实现的功能是，系统在上电的时候首先设置调制电压，即各个 D/A 转换芯片的输出电压，然后将扫描按键输入，根据按键输入来执行不同的调制输出模式。

26.4.1 系统状态编码

根据前面电路图中的接线方式，P1 口控制了整个系统的运行模式。程序中采用状态转移的方式进行控制，P1 口的控制状态的编码具有如下几种。

- 设置 D/A 转换器 1，二进制编码为 1011 1110B，相应的十六进制编码为 BEH；
- 设置 D/A 转换器 2，二进制编码为 1011 1101B，相应的十六进制编码为 BDH；
- 设置 D/A 转换器 3，二进制编码为 1011 1011B，相应的十六进制编码为 BBH；
- 调制电压 1 输出，二进制编码为 1111 0111B，相应的十六进制编码为 F7H；
- 调制电压 2 输出，二进制编码为 1110 1111B，相应的十六进制编码为 EFH；
- 调制电压 3 输出，二进制编码为 1101 1111B，相应的十六进制编码为 DFH；
- 无调制输出，二进制编码为 1011 1111B，相应的十六进制编码为 BFH。

程序中主要通过输出这些编码来实现系统控制的目的。

26.4.2 建立项目

这里采用 Keil C51 语言编写程序。首先建立项目，操作步骤如下：

- (1) 打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 26-15 所示。
- (3) 单击“确定”按钮，此时弹出“μ Vision3”对话框，如图 26-16 所示。单击“是”按钮，完成工程的建立。

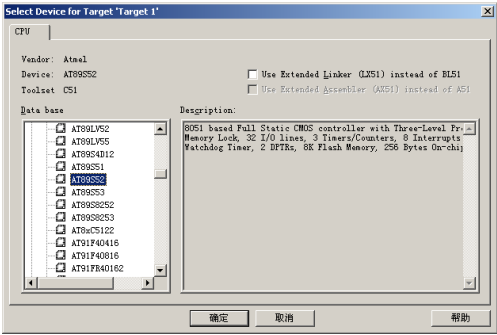


图 26-15 选择单片机 AT89S52



图 26-16 “μ Vision3”对话框

- (4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

26.4.3 主程序

主程序的流程图如图 26-17 所示。

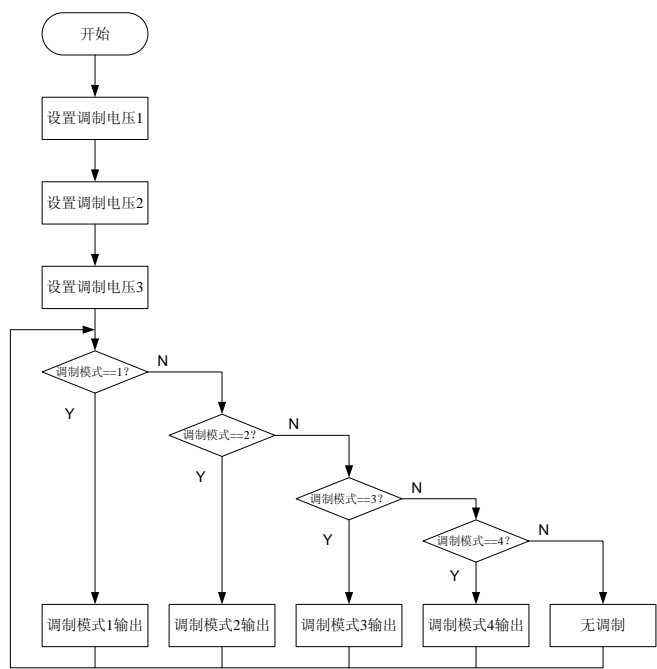


图 26-17 主程序流程图

主程序中分别设置三个调制电压，即 AD558 的模拟电压输出。然后程序中使用 while 循环来扫描键盘，根据按键输入的调制模式，分别进行各种调制模式的输出。主程序的代码示例如下：

```
#include <stdio.h> //头文件
#include <reg51.h>
#include <intrins.h>

sbit Mode1=P2^0; //调制模式 1 的输入
sbit Mode2=P2^1; //调制模式 2 的输入
sbit Mode3=P2^2; //调制模式 3 的输入
sbit Mode4=P2^3; //调制模式 4 的输入

void ModeFun0(void); //无调制模式
void ModeFun1(void); //调制模式 1
void ModeFun2(void); //调制模式 2
void ModeFun3(void); //调制模式 3
void ModeFun4(void); //调制模式 4

void main()
{
    P1=0xBF; //无参数设置，无调制
    P0=0x11; //电压 1
    P1=0xBE; //设置电压 1
    P1=0xBF; //无参数设置，无调制
    P0=0x22; //电压 2
    P1=0xBD; //设置电压 2
    P1=0xBF; //无参数设置，无调制
```

```
P0=0x33; //电压 3
P1=0xBB; //设置电压 3
P1=0xBF; //无参数设置，无调制
while(1)
{
    if(Mode1==0 && Mode2==1 && Mode3==1 && Mode4==1) //调制模式 1
    {
        ModeFun1();
    }
    else if(Mode1==1 && Mode2==0 && Mode3==1 && Mode4==1) //调制模式 2
    {
        ModeFun2();
    }
    else if(Mode1==1 && Mode2==1 && Mode3==0 && Mode4==1) //调制模式 3
    {
        ModeFun3();
    }
    else if(Mode1==1 && Mode2==1 && Mode3==1 && Mode4==0) //调制模式 4
    {
        ModeFun4();
    }
    else
    {
        P1=0xBF; //无调制
    }
}
```

在主程序中，调用了调制模式处理函数来进行处理，下面分别介绍各个调制模式处理函数。

26.4.4 无调制模式函数

无调制模式函数 ModeFun0 中控制模拟开关输出 0 电压，并通过函数_nop_来实现特定的延时用于保持输出。该函数的代码示例如下：

```
void ModeFun0(void)
{
    int i;
    P1=0xBF; //调制电压 0 输出
    _nop_(); //延时
    P1=0xBF;
    for (i=0;i<4;i++) //延时
    {
        _nop_();
    }
}
```

26.4.5 调制模式 1 函数

调制模式 1 函数 ModeFun1 中控制模拟开关输出 2.1V 电压，并通过函数_nop_来实现特定的延时用于保持输出。该函数的代码示例如下：

```
void ModeFun1(void)
{
    int i;
    P1=0xF7; //选通调制电压 1 输出
    _nop_(); //保持输出
    P1=0xBF; //无参数设置，无调制
}
```



```

        for (i=0;i<4;i++)                //保持输出
        {
            _nop_();
        }
    }

```

26.4.6 调制模式 2 函数

调制模式 2 函数 ModeFun2 中控制模拟开关输出 4.2V 电压，并通过函数_nop_来实现特定的延时用于保持输出。该函数的代码示例如下：

```

void ModeFun2(void)
{
    int i;
    P1=0xEF;                            //选通调制电压 2 输出
    _nop_();
    P1=0xBF;
    for (i=0;i<4;i++)
    {
        _nop_();
    }
}

```

26.4.7 调制模式 3 函数

调制模式 3 函数 ModeFun3 中控制模拟开关输出 6.3V 电压，并通过函数_nop_来实现特定的延时用于保持输出。该函数的代码示例如下：

```

void ModeFun3(void)
{
    int i;
    P1=0xDF;                            //选通调制电压 3 输出
    _nop_();
    P1=0xBF;
    for (i=0;i<4;i++)
    {
        _nop_();
    }
}

```

26.4.8 调制模式 4 函数

调制模式 4 函数 ModeFun4 为阶梯波调制模式，这里分别调用 ModeFun0、ModeFun1、ModeFun2 和 ModeFun3 函数，依次输出调制电压 0、调制电压 1、调制电压 2 和调制电压 3。该函数的代码示例如下：

```

void ModeFun4(void)
{
    ModeFun0();                          //调制电压 0 输出
    ModeFun1();                          //调制电压 1 输出
    ModeFun2();                          //调制电压 2 输出
    ModeFun3();                          //调制电压 3 输出
}

```

程序设计完毕后，便可以进行编译源程序。如果程序输入无误，便可以编译成功。

26.5 光通信电压调制电路实例——仿真部分

对于上一节的程序，在实际进行硬件电路设计前可以进行程序的仿真，以保证程序运行的正确性。

26.5.1 程序仿真

在程序编译通过后，便可以进行程序仿真。在 Keil μ Vision3 中，通过模拟按键输入来仿真不同调制模式的执行，通过观察各个引脚电平的变化，便可以判断程序执行的正确性。仿真的操作步骤如下：

- (1) 首先，选择“Debug”→“Start/Stop Debug Session”命令，进入仿真调试模式。
- (2) 选择“Peripherals”→“I/O-Ports”→“Port 0”命令，打开并行端口 P0 的仿真窗口。以同样的操作可以打开并行端口 P1 和并行端口 P2。
- (3) 在程序代码窗口，按 F10 键逐条执行程序，观察并行端口的电平是否按规定的时序进行变化。

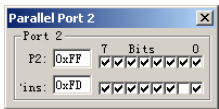


图 26-18 按键 2 按下

- (4) 在 P2 端口的仿真窗口中，将 P2.1 引脚置为低电平，仿真按键 2 按下，即选择调制模式 2，用半波电压进行调制。
- (5) 按 F10 键执行程序，在 if 语句中，可以看到程序执行进入第二个分支，执行 ModeFun2 函数。从中可以看出在调制模式 2 下，端口输出是否正确，如图 26-18 所示。

26.5.2 运行效果

在这个电路中，采用 D/A 转换芯片 AD558 实现了对要求的模拟调制电压信号的输出。通过实际测试对光脉冲进行调制，取得了很好的效果。电路实际的调制电压输出，可以在示波器上看到如图 26-19 所示波形。其中显示了在调制模式 4 下，电路输出的阶梯电压调制的情形。从图 26-19 中可以看出该电路系统的输出完全符合要求，具有很高的波形质量。

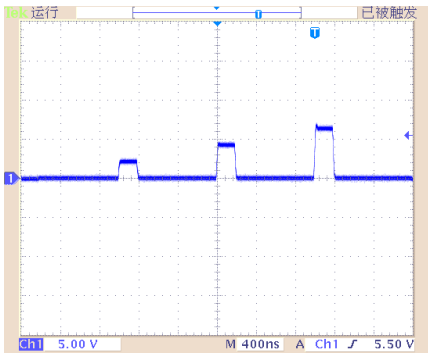


图 26-19 调制模式 4 在示波器上的输出波形

26.6 小结

本章首先详细介绍了 D/A 转换器的基本知识，包括 D/A 转换器的原理、D/A 转换器的类型及技术参数，然后介绍了一个高速易使用的 D/A 转换芯片 AD558。最后通过一个具体的实例，介绍了如何采用 AD558 在光通信领域中完成对光信号的相位精确调制。D/A 转换器在测控领域有着广泛的应用，扩展了 8051 单片机处理模拟信号的能力，是现代电子设计中不可缺少的一部分。

第 27 章 可编程逻辑器件 CPLD

可编程逻辑器件是单片机之外的另一种常用的逻辑控制单元。它是一种半定制专用集成电路（ASIC—Application Specific Integrated Circuit），其功能可由设计者根据自己的需要来加以定义。目前普遍使用的这类器件有 CPLD 和 FPGA 两种，其以工作速度快、集成度高、功耗低、适应性强等优点，受到广大电子设计人员的普遍欢迎。

CPLD/FPGA 和单片机一样，都是可重复擦写的器件，不过这两者是按照不同的原理而设计的。其性能各有优劣，有些功能两者都可以实现，但有些功能用单片机或 CPLD/FPGA 之一实现比较方便。因此，对于复杂的系统，很多时候需要两者配合使用，发挥各自的优势。本章将主要以广泛使用的 Altera 的 EPM7128SLC84-15 芯片为例进行介绍，包括程序下载及如何实现使用 CPLD 来扩展单片机 I/O 接口。

27.1 可编程逻辑器件概述

可编程逻辑器件（Programmable Logic Device）简称 PLD，是一种用户可配置的逻辑器件。其以成本低、使用灵活，设计周期短和可靠性高等优点，近年来得到快速的发展，目前成为专用集成电路的一个重要分支。

27.1.1 可编程逻辑器件的发展

可编程逻辑器件从产生到现在经历了近 40 年的发展，其结构、工艺、集成度、速度和性能等都得到不断改进和提高，现在已经形成很丰富完善的产品体系。

20 世纪 70 年代，出现了最早的可编程逻辑器件 PROM，以及可编程逻辑阵列（Programmable Logic Array，简称为 PLA）和可编程阵列逻辑（Programmable Array Logic，简称为 PAL）。

20 世纪 80 年代，PLD 进入实质性的发展阶段。Altera 推出了新型的可编程逻辑器件（Erasable Programmable Logic Device，简称为 EPLD）。同时，Xilinx 推出了现场可编程逻辑器件（Field Programmable Gate Array，简称为 FPGA）。紧接着，Lattice 又推出了复杂可编程逻辑器件（Complex PLD，简称为 CPLD）。

20 世纪 90 年代以后，随着集成电路制造工艺的发展，CPLD 和 FPGA 的集成度越来越高，规模越来越大，设计方式越来越灵活，设计软件也逐步完善。到目前为止，各大芯片厂商推出了各种类型和系列的 CPLD 和 FPGA，如 Altera 的 MAX 系列、MAX II 系列、FLEX 系列，Xilinx 的 Spartan 系列，Lattice 的 MachXO 系列等。其产品种类和应用广泛程度不亚于单片机的规模。

27.1.2 CPLD 的结构及其逻辑实现

可编程逻辑器件结构上是由与阵列和或阵列构成，原理是基于所有的逻辑电路都可以用“积之和”的布尔方程来表示。其中，两级逻辑阵列的一个阵列或全部阵列可以通过编程来实现不同的组合，从而达到要求的逻辑功能。目前应用最为广泛的是 CPLD 和 FPGA，这两者在结构上是完全不同的。下面首先介绍 CPLD 的结构及其逻辑实现。

1. CPLD 的内部结构

CPLD 内部可以看做多个 PLD 宏单元的集成，通常包括 20~256 个宏单元。另外，CPLD

内部集成了密集的布线阵列，可以实现各个宏单元及 I/O 引脚之间的连接。CPLD 的特点是，宏单元扇入大，拥有较少的触发器，引脚之间有确定的延时，可以代替一些常用的标准逻辑芯片。一般 CPLD 芯片均支持 JTAG 协议，使用相应公司提供的编译软件和下载电路，便可实现对 CPLD 芯片的程序下载。

这里以 Altera 公司的 MAX7000S 系列芯片为例，其内部结构如图 27-1 所示，其他型号的结构与此类似。从图 27-1 中可以看出，CPLD 主要有三个功能模块，即宏单元（Macrocell）、可编程连线阵列（PIA）和 I/O 控制块。

- 宏单元是 CPLD 的基本组件，所有的逻辑功能都是由它来实现的。图 27-1 中 LAB A～LAB D 是多个宏单元的集合。
- 可编程连线用于连接所有的宏单元，从而实现信号的传递。
- I/O 控制块负责各个引脚的输入和输出的电气特性控制。

另外，INPUT/GCLK1、INPUT/GCLRn、INPUT/OE1、INPUT/OE2 分别是全局时钟、清零和输出使能引脚，其他类型的 CPLD 一般都有类似功能的引脚。这几个引脚有专用连线与 CPLD 的每个宏单元相连，经过合理布局，使信号到每个宏单元的延时基本相同并且最小。

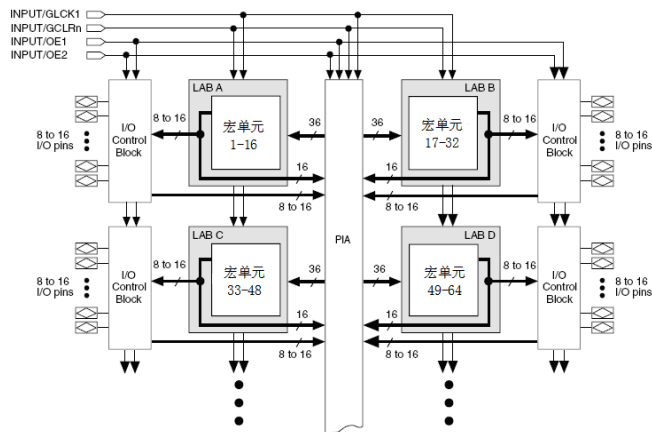


图 27-1 CPLD 的内部结构

CPLD 宏单元大多采用乘积项结构，如图 27-2 所示，其左侧便是乘积项阵列。乘积项阵列原理上是一个与或阵列，每个阵列交叉处是一个可编程的熔丝，相应熔丝导通就可以实现逻辑与的功能。后面是一个或阵列，两者一起便可以完成特定的组合逻辑。

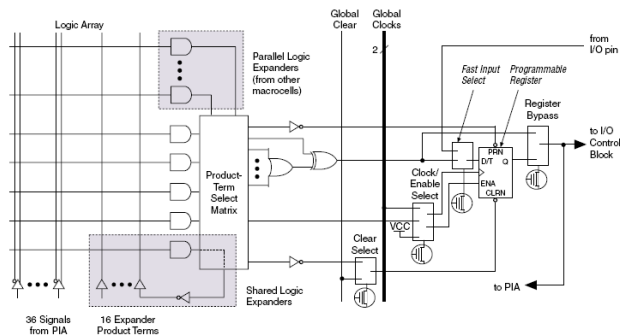


图 27-2 CPLD 的宏单元结构

在图 27-2 中，右侧的 Programmable Register 为一个可编程的 D 触发器，可以使用专用的全局时钟和全局清零，也可以使用内部组合逻辑产生的信号作为时钟和清零。如果逻辑电路中如果不需要使用触发器，可以将触发器屏蔽，信号便可以直接输出到 I/O 引脚。

2. CPLD 的逻辑实现

一个逻辑电路的实现，主要依靠 CPLD 内部乘积项结构的逻辑分解和互连来实现的。下面以一个简单的组合逻辑电路为例进行说明，如图 27-3 所示。

其中，输入端为 A、B 和 C，输出端为 F，该逻辑电路的逻辑表达式为 $F = (A + B)\overline{C} = \overline{A}\overline{C} + B\overline{C}$ 。

如果使用 CPLD 的逻辑来实现，则如图 27-4 所示。其中，A、B、C 由 CPLD 芯片的管脚输入后，进入可编程连线阵列（PIA），在 CPLD 内部会产生 A 和 \overline{A} 、B 和 \overline{B} 、C 和 \overline{C} 共 6 个信号。图 27-4 中的“叉”表示相连，对应于可编程熔丝导通。按照图 27-4 中的连接，可得 $F = F1 + F2 = \overline{A}\overline{C} + B\overline{C}$ ，这样就实现了图 27-3 所示的组合逻辑。

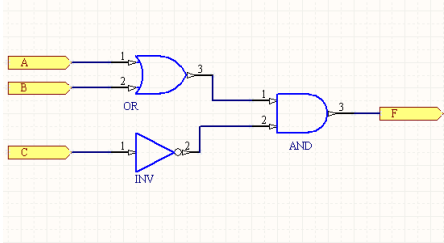


图 27-3 简单的组合逻辑电路

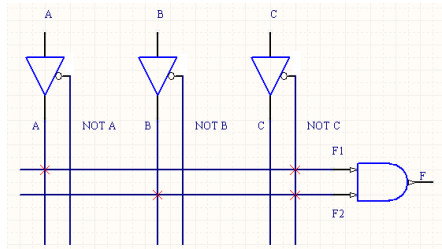


图 27-4 CPLD 的逻辑实现

这个例子比较简单，仅用一个宏单元便可以实现。但对于复杂的逻辑电路或时序电路，往往一个宏单元是不能完成的，这时就需要通过并联和共享扩展项将多个宏单元组合来使用。这样 CPLD 便可以实现更复杂的逻辑电路了。所有的这些互联的操作都是由软件自动完成的，不需要人为干预。

27.1.3 FPGA 的结构及其逻辑实现

FPGA 具有与 CPLD 完全不同的内部结构，其内部是类似于门阵列的结构，可编程的逻辑单元位于芯片的中央，并排成阵列，四周采用可编程的 I/O 焊盘连接。逻辑单元之间使用可编程的互连资源，用来实现逻辑单元及 I/O 引脚之间的联通，从而实现复杂的逻辑功能。FPGA 一般都含有成千上万的逻辑门单元，还有许多触发器。

FPGA 的逻辑单元可以分为粗粒结构和细粒结构两种。粗粒结构的 FPGA，其逻辑单元较大，一般含有两个以上的查询表和触发器，比较适合高性能方面的应用。细粒结构的则逻辑单元相对较小，功能逻辑模块和触发器都相对较少。

与 CPLD 相比，FPGA 的容量通常大于 CPLD，而且 FPGA 的逻辑单元扇入小，触发器数量比 CPLD 要多，布局和布线的延时是不确定的，但一般是可知的。

1. FPGA 的内部结构

FPGA 内部大多采用查找表结构。查找表（Look-Up-Table）简称为 LUT，其本质上是一个 RAM。现在流行的 FPGA 中都采用 4 输入的 LUT，每个 LUT 相当于一个有 4 位地址线的 RAM，内部可以保存 16 个数据结果。当用户设计了一个逻辑电路后，编译软件会自动计算该逻辑电路的所有可能输出结果，并把结果写入 RAM 的存储单元中。这样，每输入一个信号进行逻辑运算，在 FPGA 中就相当于输入一个地址，在 RAM 中进行查表，找出该地址所对应的 RAM 内容，然后输出该内容即可。

Xilinx 公司的 Spartan 系列和 Altera 公司的 FLEX、APEX、ACEX 系列等均采用查找表结构。下面以 Altera 公司的 FLEX 系列芯片为例，介绍 FPGA 的内部结构，如图 27-5 所示。其主要包括 LAB、RAM 块、I/O 块和可编程行/列连线。一个 LAB 由 8 个逻辑单元（LE）构成，每个 LE 又由一个 LUT、一个触发器和部分逻辑组合而成。LE 是 FLEX 系列 FPGA 芯片实现逻辑的最基本结构，如图 27-6 所示。

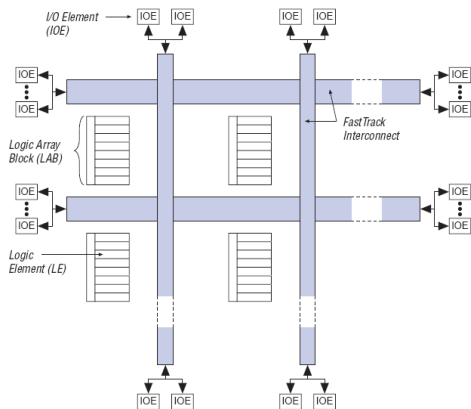


图 27-5 FLEX 系列芯片的内部结构

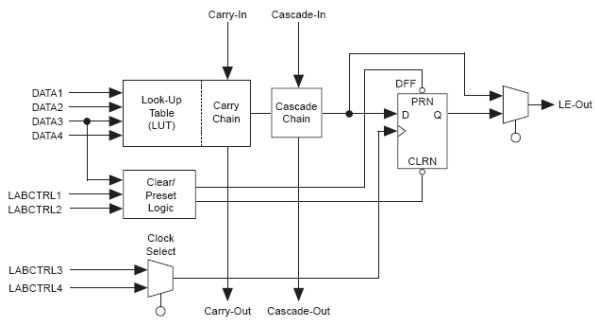


图 27-6 FLEX 的逻辑单元

2. FPGA 的逻辑实现

这里仍以图 27-3 所示的组合逻辑电路为例进行讲解。假设 A、B 和 C 由 FPGA 的 I/O 管脚输入后进入可编程连线，然后作为地址线连到 LUT。LUT 中已经事先写入了所有可能的逻辑结果，通过地址查找到相应的数据然后输出，这样组合逻辑就实现了。

如表 27-1 所示列出了该组合电路的 LUT 实现，其中输入只有三个，所以 LUT 的地址只用了三个。

表 27-1 实际逻辑电路的查找表实现

实际逻辑电路		LUT 的实现方式	
ABC 输入	逻辑输出	地 址	RAM 中存储的内容
000	0	X000	0
001	0	X001	0
010	1	X010	1
011	0	X011	0
100	1	X100	1
101	0	X101	0
110	1	X110	1
111	0	X111	0

这个电路比较简单，仅用一个 LUT 便可以完成。对于复杂的逻辑电路，一个 LUT 不能完成，可以通过进位逻辑将多个单元相连，这样 FPGA 便可以实现复杂的逻辑电路了。这些操作都是由软件自动完成的，不需要人为干预。

27.2 硬件描述语言简述

硬件描述语言（Hardware Description Language）简称为 HDL，是一种能够以形式化方式描述电子系统的硬件行为、结构和数据流的语言。硬件描述语言借鉴了其他高级语言的功能特性，对数字电路的结构和行为进行了高度抽象化、规范化的形式描述，并对设计进行不同层次的仿真和优化。

硬件描述语言是伴随着大规模集成电路的发展而形成的，至今已有 40 多年的历史，主要用来解决大规模复杂集成电路的设计问题。特别是 CPLD 和 FPGA 广泛普及后，硬件描述语言在广大电子工程师中得到广泛应用。很多 EDA 厂商和科研机构都开发了自己的电路硬件描述语言，比较有名的有 Altera 公司的 AHDL 语言，Data I/O 公司的 ABEL-HDL 语言，Xilinx 公司的 Verilog HDL 语言。由于这些硬件描述语言之间不能通用，限制了 CPLD 和 FPGA 的广泛使用。此时迫切需要建立一个统一的标准。

27.2.1 硬件描述语言 VHDL 概述

VHDL（VHSIC Hardware Description Language）语言是一种硬件描述语言，20 世纪 80 年代初，由美国国防部在其超高速集成电路计划（VHSIC）中提出。后来，该语言逐步得到推广，应用越来越广泛。1987 年 12 月，电气和电子工程师协会 IEEE 正式将 VHDL 语言作为国际标准，编号为 IEEE Std1076-1987，即 VHDL’87。1993 年，又对该版本进行的进一步的修改，新的版本号为 IEEE Std1076-1993，即 VHDL’93。

1995 年，Verilog HDL 语言也被正式定为国际标准，编号为 Verilog HDL1364-1995。这里主要介绍 VHDL 语言。

VHDL 语言专门用于对 CPLD 和 FPGA 等可编程逻辑器件进行编程，实现各种组合逻辑和时序电路等。它是一种独立于制造工艺的语言，是目前标准化程度最高的硬件描述语言。VHDL 语言的主要特点如下：

- 描述能力强，可从门级、电路级直至系统级进行描述、仿真和综合。
- 具有良好的可读性，容易被计算机处理，也容易被读者理解。
- 数据类型丰富，既支持预定义数据类型，也支持用户自定义数据类型。
- 同时拥有一般语言的顺序语句处理能力和硬件的并行行为的处理能力。
- 支持自下而上的设计，也支持自上而下的设计；既支持模块化设计，也支持层次化设计。
- 具有良好的可移植性，其作为一种 IEEE 的工业标准，被普遍接受，可以在多种 EDA 编译环境中使用，兼容性好。
- VHDL 的硬件描述与工艺技术无关，具有很长的生命周期，不会因为工艺的变化而使设计无效。

27.2.2 VHDL 程序结构

一个完整的 VHDL 程序通常包括实体(Entity)、构造体(Architecture)、配置(Configuration)、包集合(Package)和库(Library) 5 个部分。其中实体和构造体是必须的，其他几个是可选项。VHDL 语言程序的基本结构，如图 27-7 所示。

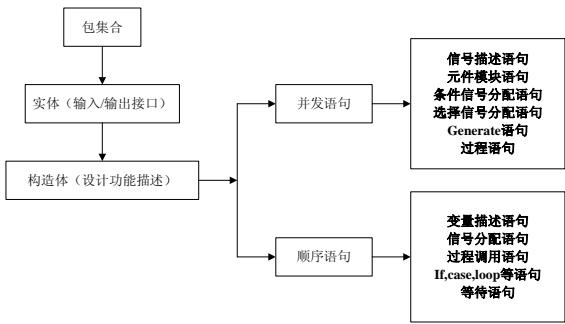


图 27-7 VHDL 语言程序的基本结构

1. 实体

实体是由实体名、类属参数声明、端口声明、实体声明和实体语句等几个部分组成。在 VHDL 语言中，实体的语法格式如下：

```
entity 实体名 is
    [类属参数声明；]
    [端口声明；]
    实体说明部分；
    [begin
        实体语句部分；]
```

```
end [entity] [实体名];
```

其中，类属参数声明用来确定所使用的局部变量的大小或实体的时限；端口声明用来确定输入/输出端口的数量及类型。带方括号“[]”的部分为可选项，每个语句都以半角分号“;”作为结束符（entity 实体名 is 除外），每行只能写一条语句。示例如下：

```
entity MyCPLD is
  port(a   :in std_logic;
        b   :in std_logic;
        c   :in std_logic;
        d   :out std_logic);
end VHDLtest;
```

在该实体声明中，包含了名称 MyCPLD 和端口声明语句 port，端口列表中定义 a、b、c 为输入端口，d 为输出端口。

2. 构造体

构造体定义了设计实体的功能，具体指明了设计实体的行为、元件及内部的连接关系。构造体必须跟在实体的下面，在构造体内可以采用如下三种方法来实现对设计实体的描述。

- 行为描述：一般采用进程语句，顺序描述设计实体的行为，又称为数学模型描述。
- 数据流描述：一般采用进程语句，顺序描述数据流在控制流作用下被加工、处理、存储的全过程、又称为寄存器传输描述。
- 结构描述：一般采用并行处理语句描述设计实体内的结构组织和元件互连关系，又称为逻辑元件连接关系。

不同的描述方式，只体现在所使用描述语句上，而构造体的结构则完全一样，构造体的一般书写格式如下：

```
architecture 构造体名 of 实体名 is
    [定义语句, ]内部信号, 常数, 数据类型, 函数等的定义
Begin
    [并行处理语句];
    [进程语句];
end 构造体名;
```

其中，“architecture 构造体名 of 实体名 is”指名了构造体名，architecture 为关键字；Begin 为构造体的开始语句，构造体的内容放在保留字 begin 与 end 之间，在构造体中的所有语句是同时执行的。示例如下：

```
architecture Arch of Test is
begin
    c<=a and b;
    d<=a xor b;
    e<=a or b;
end Arch;
```

在该 VHDL 程序中，Test 为实体名，Arch 为构造体名。程序中并行执行三个逻辑运算语句。下面举例说明一个典型的 4 位并入串出的移位寄存器 VHDL 程序，如图 27-8 所示，其 VHDL 语言程序示例如下：

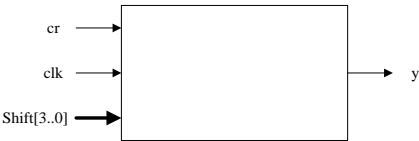


图 27-8 4 位并入串出移存器

```
library ieee;
use ieee.std_logic_1164.all;
```

--库参考


```

use ieee.std_logic_unsigned.all;

entity VHDLtest is                                --实体说明
    port(clk :in std_logic;                        --端口说明, clk 输入
          cr  :in std_logic;                      --cr 输入
          shift :in std_logic_vector(3 downto 0); --shift 输入
          y    :out std_logic);                  --y 输出
end VHDLtest;

architecture Arch of VHDLtest is                --构造体声明
    signal k:std_logic_vector(3 downto 0);       --定义内部信号
    signal q:std_logic_vector(1 downto 0);
begin
    P1:process(clk)                             --进程 1
    begin
        if(clk'event and clk='1') then          --if 语句
            q<=q+1;
        end if;
    end process;
    P2:process(clk)                             --进程 2
    begin
        if cr='0' then                          --if 语句
            elsif(clk'event and clk='1') then
                if q>"00" then
                    k(3 downto 1)<=k(2 downto 0); --移位
                elsif q="00" then
                    k<=shift;
                end if;
            end if;
            y<=k(3);
        end process;                          --进程结束
    end Arch;

```

上面这段 VHDL 程序中, 使用 if 语句判断 clk 的上升沿, 将 4 位的并行输入信号逐位从 y 端口串行输出, cr 用于清零。

27.3 Altera 常用 CPLD 芯片介绍

Altera 公司的 MAX7000 系列是使用比较广泛的 CPLD, 下面介绍其中的芯片 EPM7128SLC84-15, 这是基于乘积项结构的可编程逻辑器件, 其中有 128 个宏单元, 8 个逻辑阵列块。该芯片有 64 个引脚可以用做通用输入输出引脚, 对外扩展能力强大。引脚到引脚之间的延时快达 5ns, 计数器频率高达 175.4MHz。

1. 引脚分配

EPM7128SLC84-15 的 PLCC 封装的引脚分配图, 如图 27-9 所示。EPM7128SLC84-15 共有 84 根引脚, 其功能分别介绍如下所述。

- VCCIO (Pin13、Pin26、Pin38、Pin53、Pin66、Pin78): 为 CPLD 的 I/O 引脚电源电平, 为 5V 或 3.3V。
- VCCINT (Pin3、Pin43): 为 CPLD 的内部操作电源电平, 为 5V。
- GND (Pin7、Pin19、Pin32、Pin42、Pin47、Pin59、Pin72、Pin82): 为芯片的电源的地线。
- INPUT/GCLK1 (Pin83): 为 CPLD 的全局时钟。

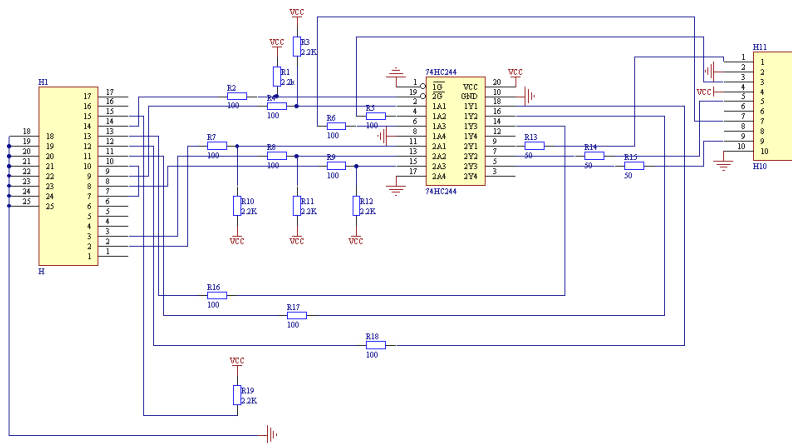


图 27-11 ByteBlasterMV 型并口下载电路原理图

表 27-2 元器件列表

元 器 件	数 值	数 量
H1	25 针并口	1 个
H11	5×2 的接插件	1 个
74HC244	74HC244	1 个
R2、R4~R9、R16~R18	100 Ω	10 个
R13~R15	50 Ω	3 个
R1、R3、R10~R12、R19	2.2k Ω	6 个



图 27-12 下载线实物

在使用时，只需将下载电路的一端连接计算机并口，另一端中的（VCC、GND、TDI、TMS、TCK、TDO）连接到目标电路板中 CPLD 芯片的相应引脚即可。当目标系统上电后，编译程序便可以检测到下载电路，后面便可以将编译通过的程序下载到芯片内部执行。

该下载电路所使用的元件少，结构简单，可以将其装入并口连接器中。这样，可以减小电路的体积，使用起来也很方便。最后完成的 VHDL 程序下载线，如图 27-12 所示。

27.4 使用 CPLD 扩展 51 单片机 I/O 接口

可编程逻辑器件 CPLD/FPGA 的主要优点是内部逻辑资源丰富，输入输出接口多，非常适合于逻辑电路及有一定时序要求的电路。下面以前面介绍的 Altera 公司的 CPLD 芯片 EPM7128SLC84-15 为例，讲解 CPLD 的程序设计，以及如何扩展单片机的接口。

27.4.1 CPLD 扩展单片机 I/O 接口原理

8051 系列单片机的 I/O 端口比较少，当系统外扩存储器，或者使用串口及外部中断资源后，剩余可用的 I/O 端口便更少了。为此，经常需要进行单片机 I/O 接口的扩展。

前面介绍过，可以使用外部 RAM 地址空间及串口的模式 0 来实现扩展的 I/O 端口。下面介绍使用可编程逻辑器件 CPLD 丰富的 I/O 引脚资源，来扩展单片机 I/O 接口。

使用 CPLD 扩展单片机 I/O 接口的示意图，如图 27-13 所示，其中扩展了 4 个输出接口。也可以采用同样的方法扩展输入接口。整个系统涉及单片机的程序设计、CPLD 的程序设计等，下面分别进行介绍。

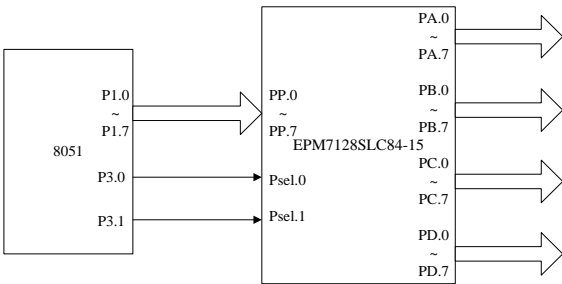


图 27-13 CPLD 扩展单片机 I/O 接口示意图

27.4.2 电路图

这里采用层次式电路设计，层次式电路的母原理图，如图 27-14 所示。在母原理图中，定义了两部分的接口，单片机和 CPLD 之间的接口通过 P1 端口和 P3.0、P3.1 引脚连接，P1 端口输出并行数据，P3.0 和 P3.1 用于选择输出的扩展并行口。

- 当 P3.0=0 且 P3.1=0 时，选择 PA 口输出；
- 当 P3.0=1 且 P3.1=0 时，选择 PB 口输出；
- 当 P3.0=0 且 P3.1=1 时，选择 PC 口输出；
- 当 P3.0=1 且 P3.1=1 时，选择 PD 口输出。

母原理图中有两个子原理图文件，MCU 和 CPLD。MCU 子原理图，如图 27-15 所示；CPLD 子原理图，如图 27-16 所示。

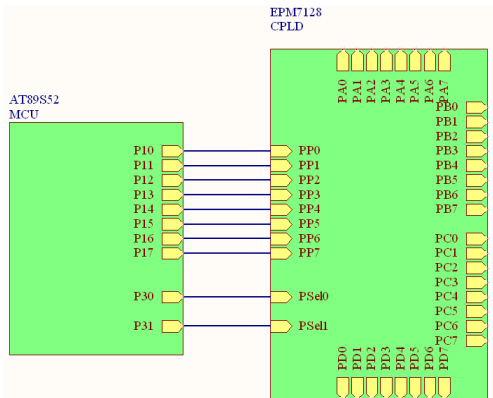


图 27-14 母原理图

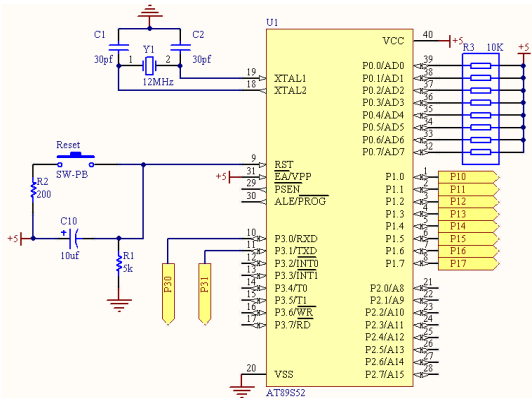


图 27-15 MCU 子原理图

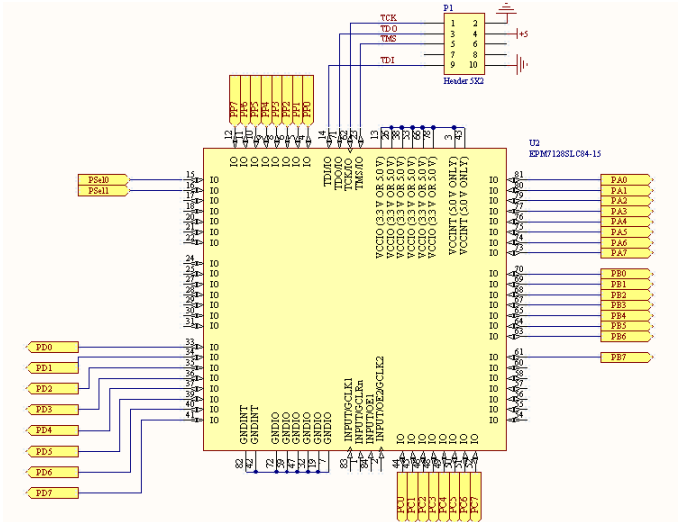


图 27-16 CPLD 子原理图

整个电路中所使用的元器件列表，如表 27-3 所示。

表 27-3 元器件列表

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89S52	1 个	R1	5k Ω	1 个
U2	EPM7128SLC84-15	1 个	R2	200 Ω	1 个
Y1	12MHz	1 个	R3	10k Ω 排阻	1 个
C1、C2	30pf	2 个	C10	10 μ f 电解电容	1 个
SW-PB	按键开关	1 个	P1	5 \times 2 接插件	1 个

在 CPLD 子原理图中，P1 留作程序下载使用，可以使用前面的 Altera 下载电路进行程序下载。

27.5 单片机程序设计

这里采用 Atmel 公司的 AT89S52 单片机，使用 Keil C51 语言编写单片机程序。

27.5.1 项目建立

整个操作在 Keil μ Vision3 编译环境中进行，具体操作步骤如下：

- （1）首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。
- （2）在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 27-17 所示。
- （3）单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 27-18 所示。单击“是”按钮，完成工程的建立。
- （4）选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件。可以在其中输入程序代码。

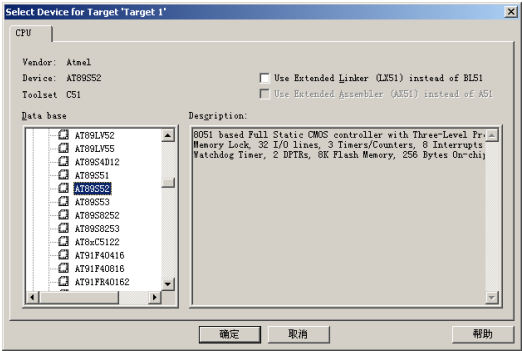


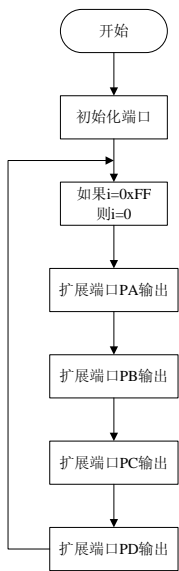
图 27-17 选择单片机 AT89S52



图 27-18 “μ Vision3”对话框

27.5.2 主程序

AT89S52 主程序的流程图，如图 27-19 所示。程序代码示例如下：



```
#include <reg52.h> //头文件
#include <stdio.h>

void main() //主函数
{
    int dataout=0; //定义并初始化整型变量
    P1=0x0; //并行端口初始化
    P3=0x0; //扩展端口选择初始化
    while (1) //主循环
    {
        if (dataout ==0xFF) //如果 dataout ==255，则
            dataout =0; //dataout =0
        P1= dataout ++; //扩展端口 PA 输出
        P3=0x0;
        P1= dataout ++; //扩展端口 PB 输出
        P3=0x01;
        P1= dataout ++; //扩展端口 PC 输出
        P3=0x02;
        P1= dataout ++; //扩展端口 PD 输出
        P3=0x03;
    }
}
```

图 27-19 主程序流程图

该程序中，首先初始化 P1 和 P3 端口。接着在 while 语句中，循环将数据传送给 CPLD 的扩展端口 PA、PB、PC 和 PD。

27.6 VHDL 程序设计

Altera 提供了优秀的 CPLD 和 FPGA 编译仿真软件，如 MAX+PLUS II 和 Quartus II，可以仿真和编译 Altera 公司的所有产品。这里采用 Altera 公司最新的 Quartus II 软件进行 VHDL 程序设计。

27.6.1 项目建立

首先，在 Quartus II 软件中建立 VHDL 设计项目，操作步骤如下：

- (1) 打开 Quartus II 软件，其界面如图 27-20 所示。
- (2) 选择“File”→“New Project Wizard”命令，打开“New Project Wizard”对话框，如图 27-21 所示。下面根据向导一步一步创建工程项目。
- (3) 单击“Next”按钮，进入文件夹设置对话框，如图 27-22 所示。其中，可以分别指定该工程项目的保存路径、工程名称和顶级设计实体的名称。
- (4) 单击“Next”按钮，进入“Add Files”对话框，如图 27-23 所示。如果以有设计源文件，可以在其中导入。这里从头建立所有的设计文件，因此不用选择，可以跳过。

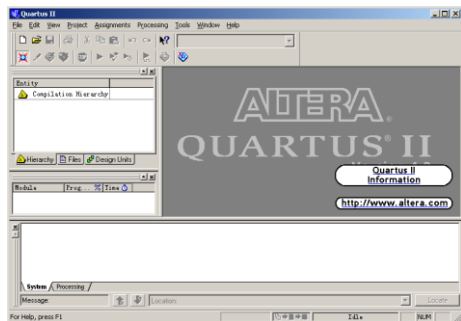


图 27-20 Quartus II 软件界面

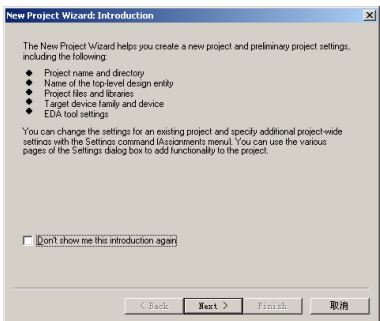


图 27-21 “New Project Wizard” 对话框

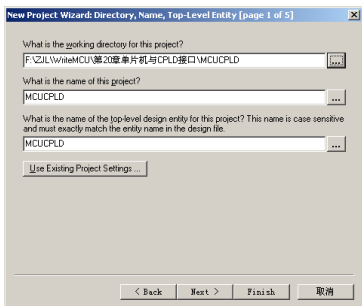


图 27-22 文件夹设置对话框

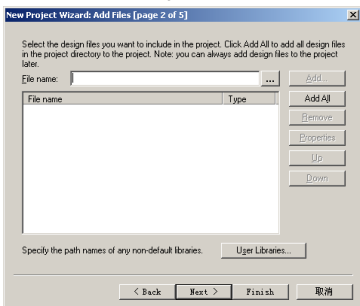


图 27-23 “Add Files” 对话框

(5) 单击“Next”按钮，进入“Family & Device Settings”对话框，如图 27-24 所示。从中选择 Family 为 MAX7000S 系列，在 Available devices 中选择 EMP7128SLC84-15 即可。

(6) 单击“Next”按钮，进入“EDA Tool Settings”对话框，如图 27-25 所示。其中，可以选择 Quartus II 软件之外提供的 EDA 工具。这里仅使用 Quartus II 软件中的编译仿真工具即可，因此，不做任何选择，直接跳过即可。

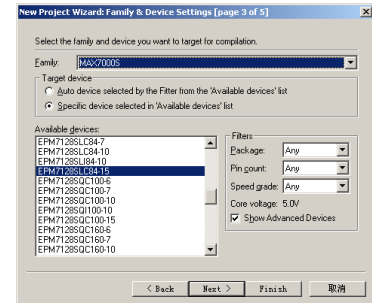


图 27-24 “Family & Device Settings” 对话框

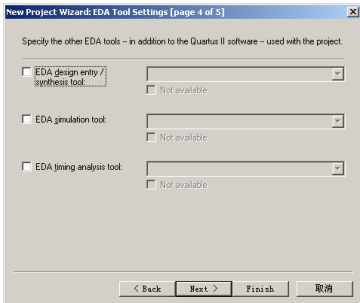


图 27-25 “EDA Tool Settings” 对话框

(7) 单击“Next”按钮，进入“Summary”对话框，如图 27-26 所示。该对话框总结了前面所有步骤对整个工程项目的设置。单击“Finish”按钮完成工程项目的建立，或者单击“Back”按钮，返回前面的步骤进行修改。

至此，便完成了整个工程项目的建立，下面便可以开始具体的程序设计。

27.6.2 程序设计

前面在 Quartus II 软件中完成了整个工程项目的建立，这里添加设计源文件并采用 VHDL 语言进行程序设计。操作步骤如下：

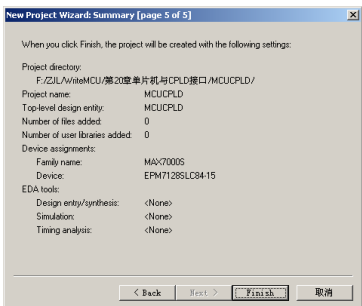


图 27-26 “Summary” 对话框

51 单片机开发与应用技术详解

(1) 在 Quartus II 软件中，选择“File”→“New”命令，弹出“New”对话框，如图 27-27 所示。在“Device Design Files”选项卡中选择“VHDL File”。

(2) 单击“OK”按钮，便可以创建一个 VHDL 源文件。

(3) 在工具栏中单击“保存”按钮，弹出“保存为”对话框，如图 27-28 所示。程序按默认的文件名保存，并同时将其添加到刚建立的工程项目中。

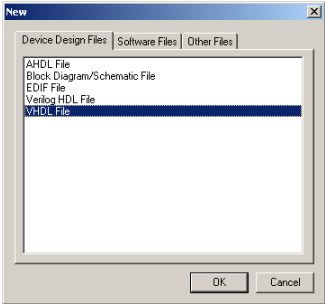


图 27-27 “New”对话框

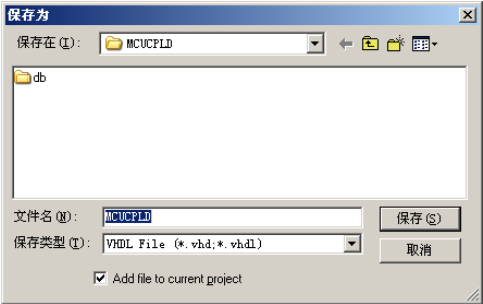


图 27-28 “保存为”对话框

(4) 此时，便可以在新建的 VHDL 设计文档中输入 VHDL 源程序。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity MCUCPLD is
    port (PP      :in std_logic_vector(7 downto 0);
          PSel    :in std_logic_vector(1 downto 0);
          PA      :out std_logic_vector(7 downto 0);
          PB      :out std_logic_vector(7 downto 0);
          PC      :out std_logic_vector(7 downto 0);
          PD      :out std_logic_vector(7 downto 0));
end MCUCPLD;

architecture Arch of MCUCPLD is
    signal SA:std_logic_vector(7 downto 0);
    signal SB:std_logic_vector(7 downto 0);
    signal SC:std_logic_vector(7 downto 0);
    signal SD:std_logic_vector(7 downto 0);

begin
    process (PSel,PP)
    begin
        if PSel="00" then
            SA<=PP;
        elsif PSel="01" then
            SB<=PP;
        elsif PSel="10" then
            SC<=PP;
        else
            SD<=PP;
        end if;
        PA<=SA;
        PB<=SB;
        PC<=SC;
    end process;
end Arch;
```

--库参考

--设计实体

--端口说明，输入端口

--扩展端口选择

--扩展输出端口 PA

--扩展输出端口 PB

--扩展输出端口 PC

--扩展输出端口 PD

--构造体

--内部变量

--过程语句

--选择端口 PA 输出

--选择端口 PB 输出

--选择端口 PC 输出

--选择端口 PD 输出

--端口输出


```
PD<=SD;
end process;
end Arch;
```

(5) 程序编写完毕后，便可以进行程序编译。

27.7 程序仿真

Altera 公司最新的 Quartus II 软件提供了良好的编译仿真操作。在进行仿真之前需要首先分配 CPLD 引脚，并确保 VHDL 程序编译成功。

27.7.1 设计 CPLD 引脚

VHDL 程序是独立于器件的，其只描述了各个输入输出端口及内部的逻辑关系。VHDL 程序的运行，还需要将每个输入输出端口映射到芯片的具体引脚。Altera 公司的 Quartus II 软件提供了简单方便的芯片引脚配置功能，支持 Altera 公司的所有 CPLD 和 FPGA 芯片。这里参照前面的电路图，对本设计中的 EPM7128SLC84-15 进行引脚分配，具体操作步骤如下：

(1) 在 Quartus II 软件中，选择“Assignments”→“Pins”命令，在软件右侧工作区出现“Assignments Editor”窗口，在其中可以进行引脚分配，如图 27-29 所示。

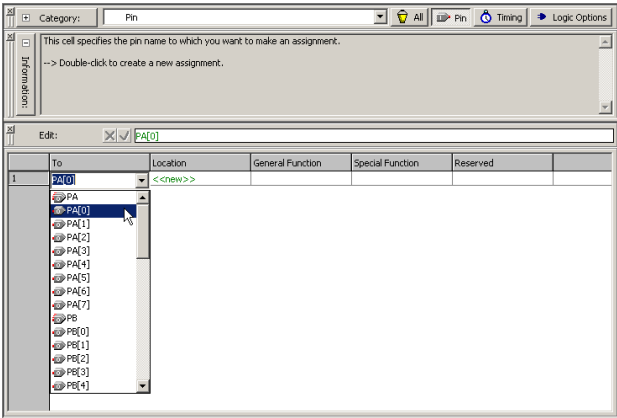


图 27-29 “Assignments Editor”窗口

(2) 在“To”栏目中可以选择该 VHDL 设计中的输入输出引脚，在“Location”栏目中可以选择需要映射到的芯片 I/O 引脚。根据前面的电路图，逐一将各个端口映射到芯片引脚上，如表 27-4 所示。

表 27-4 引脚分配

端口名称	引 脚	端口名称	引 脚
PA[0]	Pin81	PC[5]	Pin 50
PA[1]	Pin 80	PC[6]	Pin 51
PA[2]	Pin 79	PC[7]	Pin 52
PA[3]	Pin 77	PD[0]	Pin 33
PA[4]	Pin 76	PD[1]	Pin 34
PA[5]	Pin 75	PD[2]	Pin 35
PA[6]	Pin 74	PD[3]	Pin 36
PA[7]	Pin 73	PD[4]	Pin 37
PB[0]	Pin 70	PD[5]	Pin 39
PB[1]	Pin 69	PD[6]	Pin 40

续表

端口名称	引 脚	端口名称	引 脚
------	-----	------	-----

PB[2]	Pin 68	PD[7]	Pin 41
PB[3]	Pin 67	PP[0]	Pin 4
PB[4]	Pin 65	PP[1]	Pin 5
PB[5]	Pin 64	PP[2]	Pin 6
PB[6]	Pin 63	PP[3]	Pin 8
PB[7]	Pin 61	PP[4]	Pin 9
PC[0]	Pin 44	PP[5]	Pin 10
PC[1]	Pin 45	PP[6]	Pin 11
PC[2]	Pin 46	PP[7]	Pin 12
PC[3]	Pin 48	PSel[0]	Pin 15
PC[4]	Pin 49	PSel[1]	Pin 16

27.7.2 仿真操作

Altera 公司的 Quartus II 软件能够对 VHDL 程序及芯片进行精确的仿真，包括信号的延时、时序等。当完成引脚分配及程序编译后，便可以开始程序的仿真，下面具体介绍操作步骤。

(1) 在 Quartus II 软件中，选择“File”→“New”命令，弹出“New”窗口，如图 27-30 所示。在“Other Files”标签页中，选择“Vector Waveform File”选项。

(2) 单击“OK”按钮，创建波形仿真文件，如图 27-31 所示。

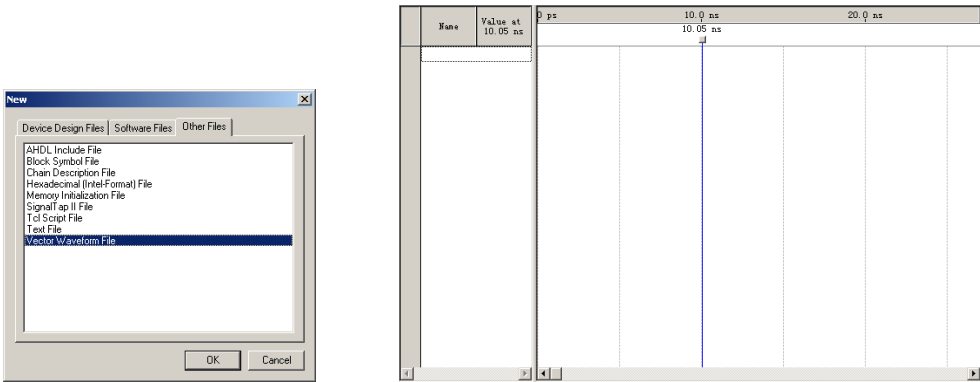


图 27-30 “New”对话框

图 27-31 波形仿真文件

(3) 对于刚建立的波形仿真文件，其中没有任何仿真变量，需要手动添加。在仿真窗口的左侧，右键单击工作区，选择“Insert Node or Bus”命令，弹出“Insert Node or Bus”对话框，如图 27-32 所示。

(4) 单击“Node Finder”按钮，弹出“Node Finder”对话框，如图 27-33 所示。

(5) 在“Filter”栏中，选择“Pins: all”，单击“List”按钮，此时，在“Node Found”栏中列出所有的引脚。选择 PA、PB、PC、PD、PP 和 PSel 等 6 个数组结构的端口，添加到右侧的“Selected Nodes”中。

(6) 单击“OK”按钮，返回“Insert Node or Bus”对话框，如图 27-34 所示。

(7) 选择“Edit”→“End Time”命令，弹出“End Time”对话框，如图 27-35 所示。在其中输入 100μs 的仿真时间。

(8) 添加完仿真变量之后的仿真窗口，如图 27-36 所示。

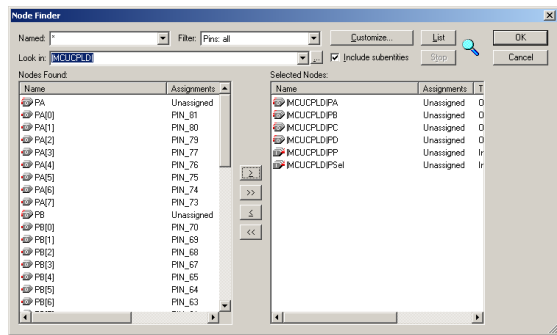
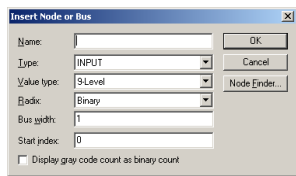


图 27-32 “Insert Node or Bus” 对话框

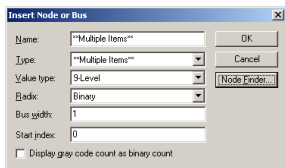


图 27-34 “Insert Node or Bus” 对话框

图 27-35 “End Time” 对话框

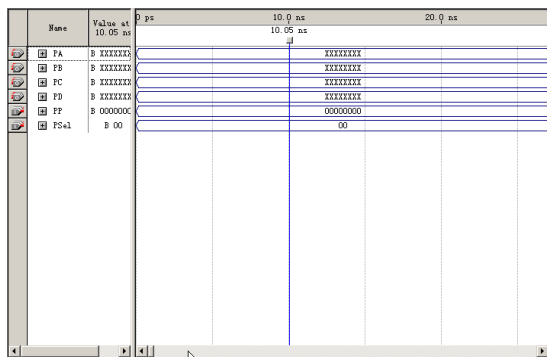


图 27-36 仿真窗口

- (9) 将输入端口 PP 和 PSel 按照如图 27-37 所示，进行编辑，即指定仿真输入信号。
- (10) 选择 “Processing” → “Start Simulation” 命令，执行仿真。仿真结束后，仿真波形如图 27-38 所示。

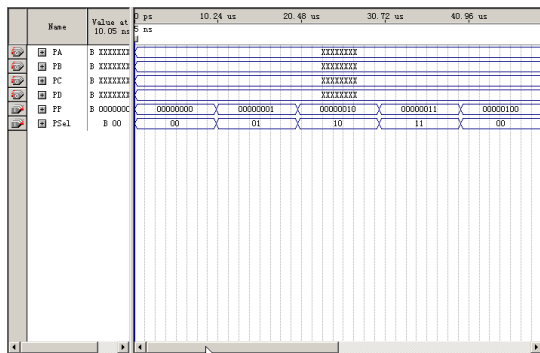


图 27-37 编辑仿真输入信号

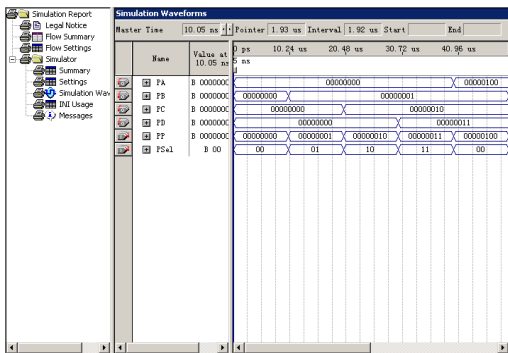


图 27-38 仿真波形

从图 27-38 中可以看出，在输入端 PSel 的选择下，轮流将 PP 端口输入的数据输出到 PA、PB、PC 和 PD 端口。仿真结果表明程序的正确性。

27.8 程序下载

在 Quartus II 软件中，可以利用前面介绍的下载电路，来将程序下载到 CPLD 芯片中执行。程序下载的过程如下：

- (1) 在 Quartus II 软件中，选择 “Tools” → “Programmer” 命令，在工作区的右侧出现程

51 单片机开发与应用技术详解

序下载界面，如图 27-39 所示。在其中可以进行编程操作。

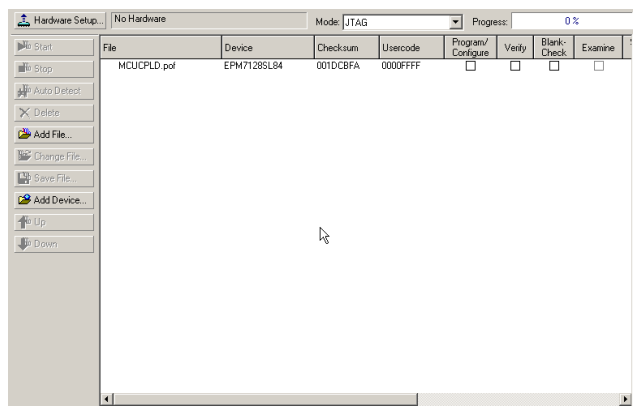


图 27-39 程序下载

(2) 如果是第一次使用，需要首先安装下载电路的驱动。单击“Hardware Setup”按钮，弹出“Hardware Setup”对话框，如图 27-40 所示。

(3)单击“Add Hardware”按钮，弹出“Add Hardware”对话框，如图 27-41 所示。在“Hardware type”选项中，选择“ByteBlasterMV or ByteBlaster II”，在“Port”中选择下载电路所连接的并行端口号。

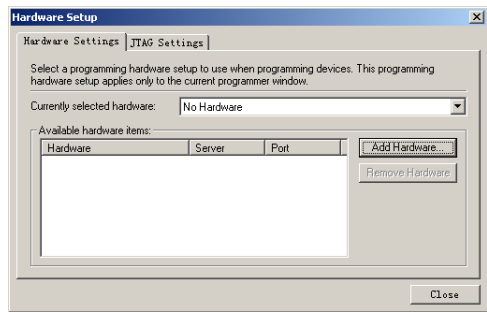


图 27-40 “Hardware Setup”对话框

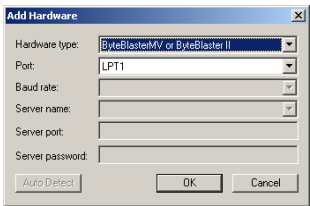


图 27-41 “Add Hardware”对话框

(4) 单击“OK”按钮，返回“Hardware Setup”对话框，如图 27-42 所示。可以看到已经将下载电路的驱动程序安装。

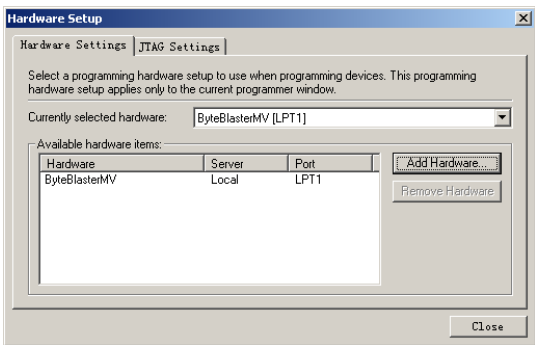
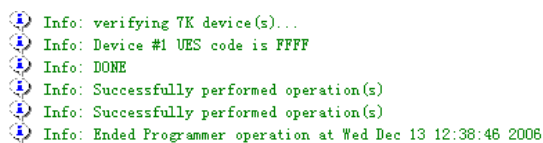


图 27-42 “Hardware Setup”对话框

(5) 单击“Close”按钮，即可退出。

(6) 下载电路的驱动安装完毕后，将“Program/Configure”、“Verify”和“Blank Check”打钩选择，便可以单击“Start”执行程序下载。

当程序下载完毕的时候，信息栏会提示操作成功，如图 27-43 所示。



```
Info: verifying TK device(s)...\nInfo: Device #1 UES code is FFFF\nInfo: DONE\nInfo: Successfully performed operation(s)\nInfo: Successfully performed operation(s)\nInfo: Ended Programmer operation at Wed Dec 13 12:38:46 2006
```

图 27-43 提示信息

通过本实例可以看出，使用 CPLD 完美地实现了扩展 4 个单片机输出端口的目的。这种方法程序设计简单，而且 CPLD 的 I/O 引脚丰富，可以使用单个芯片来扩展多个 I/O 端口，十分方便。

27.9 小结

本章首先介绍了可编程逻辑器件的发展，以及 CPLD 和 FPGA 的结构及逻辑实现。可编程逻辑器件一般采用 VHDL 语言进行设计，本章对 VHDL 语言进行了简要的介绍。本章对常用的 CPLD 芯片 EPM7128SLC84 进行展开讲解，提供了 Altera 公司的 CPLD 下载电路。本章实例部分，使用 CPLD 配合 AT89S52 来扩展了 8051 单片机的并行 I/O 端口。单片机和可编程逻辑器件内部结构不同，各有优势，实际系统中经常需要将两者结合使用。

第 28 章 51 系列单片机读写 I2C 总线

I2C 总线是 Philips 公司推出的一种双向二线制总线，全称为芯片间总线（Inter Integrate Circuit BUS）。其在芯片间使用两根连线实现全双工同步数据传送，一条数据线（SDA）和一条串行时钟线（SCL），可以很方便地构成外围器件扩展系统。

I2C 总线是很简单方便的芯片间串行扩展总线。使用 I2C 总线可以直接和具有 I2C 总线接口的单片机通信，也可以和各种类型的外围器件进行通信，如存储器、A/D、D/A、键盘、LCD 等。目前 Philips、Atmel、MAXIM 及其他集成电路制造商推出了很多基于 I2C 总线的单片机和外围器件，如 24 系列 E2PROM、串行实时时钟芯片 DS1302、USB2.0 芯片 CY7C68013A 等。

本章将主要介绍 I2C 总线的工作原理、结构及寻址方式，并重点介绍了数据传输协议及程序实现。这些程序均以子程序的形式提供，便于读者调用。最后通过具体的实例，介绍如何使用单片机读写具有 I2C 总线接口的 E2PROM。

28.1 I2C 总线概述

I2C 总线对数据通信进行了严格的定义，要进行 I2C 总线的接口设计，就需要首先了解 I2C 总线的工作原理图、寻址方式和数据传输协议等。

28.1.1 I2C 总线工作原理

典型的 I2C 总线系统结构，如图 28-1 所示。其采用两线制，由数据线 SDA 和时钟线 SCL 构成。总线上挂接的单片机（主器件）或外围器件（从器件），其接口电路都应具有 I2C 总线通信能力。

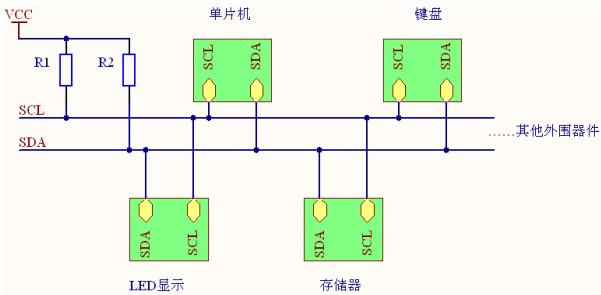


图 28-1 典型的 I2C 总线应用系统结构

I2C 总线上的数据信号完全与时钟同步，为同步传输串行总线结构。其数据传输采用主从方式，即主器件（主控制器）寻址从器件（被控制器），启动总线数据传输，并产生时钟脉冲。总线传输中的所有状态及操作都有相应的编码，主器件依照这些协议编码自动地进行总线控制与管理。被寻址的器件，即从器件，接收主器件的请求并应答。数据传输结束后，主从器

件将释放总线。

当总线空闲时，SCL 和 SDA 均为高电平。连到总线上器件的输出端口必须是漏极开路，如果任一器件输出低电平，都将使该总线的信号变低，即总线 SCL 及 SDA 上的信号都是线“与”的关系。

I2C 总线协议允许总线接入多个器件，总线中的器件即可作为主控制器也可作为被控制器，既可以是发送器，也可以是接收器。I2C 总线在进行数据交换时，作为主控制器的器件需通过总线竞争获得主控权，然后才可以启动数据传输。系统中每个器件都具有唯一的芯片地址，数据传输时通过寻址可以确定数据接收方。

I2C 总线不允许同时存在多个主器件。如果有几个主器件同时企图启动总线传送数据，I2C 总线要通过总线裁决，以决定由哪一个主器件控制总线，这样避免了数据传输产生混乱。总线裁决的机制是，当一个主器件送“1”，而另一个（或多个）送“0”，送“1”的主器件则退出竞争。在总线竞争过程中，时钟信号是各个主器件产生的异步时钟信号线“与”的结果。

I2C 总线上的主控制器，可以是带有 I2C 总线接口的单片机或其他类型微控制器，也可以不带有 I2C 总线接口，而采用软件模拟 I2C 总线的接口功能。典型的 8051 单片机没有 I2C 总线接口，一般都是通过软件模拟 I2C 协议来通信的。I2C 总线上的被控制器必须带有 I2C 总线接口。

28.1.2 I2C 总线的电气结构和负载能力

I2C 总线的 SCL 和 SDA 端口输出为漏极开路，因此使用时必须连接上拉电阻。不同型号的器件对上拉电阻的要求不同，可参考具体器件的数据手册。上拉电阻的大小与电源电压、传输速率等有关系。

I2C 总线的传输速率可以支持 100kHz 和 400kHz 两种，对于 100kHz 的速率一般采用 10KΩ 的上拉电阻，对于 400kHz 的速率一般采用 2KΩ 的上拉电阻。

I2C 总线上的外围扩展器件都是属于电压型负载的 CMOS 器件，因此总线上的器件数量不是由电流负载能力决定，而是由电容负载能力确定。I2C 总线上每一个节点器件的接口都有一定的等效电容，这会造成信号传输的延迟。通常 I2C 总线的负载能力为 400pF（通过驱动扩展可达 4000pF），据此可计算出总线长度及连接器件的数量。

另外总线上每个外围器件都有一个地址，扩展外围器件时也要受到器件地址空间的限制。

28.1.3 I2C 总线器件的寻址方式

I2C 总线上的所有器件连接在一个公共的总线上，因此，主器件在进行数据传输前选择需要通信的从器件，即进行总线寻址。

I2C 总线上所有外围器件都需要有唯一的地址，由器件地址和引脚地址两部分组成，共 7 位。器件地址是 I2C 器件固有的地址编码，器件出厂时就已经给定，不可更改。引脚地址是由 I2C 总线外围器件的地址引脚（A2，A1，A0）决定，根据其在电路中接电源正极、接地或悬空的不同，形成不同的地址代码。引脚地址数也决定了同一种器件可接入总线的最大数目。

地址位与一个方向位共同构成 I2C 总线器件寻址字节。寻址字节的格式如表 28-1 所示。方向位（R/ \overline{W} ）规定了总线上的主器件与外围器件（从器件）的数据传输方向。当方向位 R/ \overline{W} =1，表示主器件读取从器件中的数据；R/ \overline{W} =0，表示主器件向从器件发送数据。

表 28-1 寻址字节的格式

位 序	D7	D6	D5	D4	D3	D2	D1	D0
定 义	器件地址				引脚地址			方向位
	DA3	DA2	DA1	DA0	A2	A1	A0	R/ \overline{W}

28.2 I2C 总线数据传输协议及其程序详解

I2C 总线规定了严格的数据通信格式，所有具有 I2C 总线接口的器件都必须遵守。然而应

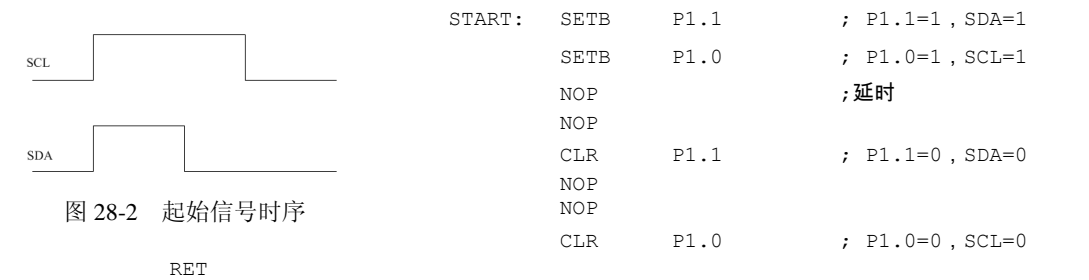
用最广的 51 系列单片机，却没有提供 I2C 总线接口。实际上，利用这些单片机的普通 I/O 口，采用软件模拟 I2C 总线 SCL 和 SDA 上的数据传送时序，完全可以实现对 I2C 总线器件的读、写操作。

下面就分别介绍 I2C 总线数据传输过程中的格式及如何使用 8051 单片机来实现数据传输。这里假设 51 系列单片机的外接晶振频率为 6MHz，单片机的机器周期为 2μs，采用 P1.0 作为时钟线 SCL，P1.1 作为数据线 SDA。

28.2.1 起始信号

起始信号用于开始 I2C 总线通信。在时钟线 SCL 为高电平期间，数据线 SDA 上出现由高电平向低电平变化的下降沿时，被认为是起始信号。起始信号出现以后，才可以进行寻址或数据传输等。起始信号的时序，如图 28-2 所示。

如果采用汇编语言进行程序设计，则其程序示例如下：



该程序中，使用 P1.1 作为 SDA，P1.0 作为 SCL，通过 SETB 指令和 CLR 指令来实现起始信号的时序。NOP 指令用于延时，使其满足传输速度的要求。

如果采用 C51 语言进行程序设计，首先需要声明 SCL 和 SDA 所使用的引脚。其声明如下：

```
sbit SCL=P1^0;
sbit SDA=P1^1;
```

这里指定 P1.0 引脚作为 SCL，P1.1 引脚作为 SDA，这在后面的子程序中同样需要用到。

```
void Start()                                //起始信号子程序
{
    SDA=1;
    DelayMs(1);                             //延时
    SCL=1;
    DelayMs(1);
    SDA=0;
    DelayMs(1);
    SCL=0;
    DelayMs(1);
}
```

该程序中，通过直接为 SDA 和 SCL 赋值来实现起始信号的时序。其中的 DelayMs 函数为延时子程序，用于满足传输速度的要求。

下面给出一个典型的延时子函数。当单片机的工作频率比较高的时候，为了保证 I2C 总线的传输速率（100kHz 或 400kHz），用于适当的延时处理。用户可以根据需要使用。

```
sbit DOG=P3^7;                             //用于显示延时子程序在执行
void DelayMs(unsigned int number)           //延时子程序，延时大小取决于工作频率和 number 值
{
    unsigned char temp;
    for(;number!=0;number--,DOG=!DOG)       //循环，DOG 用于输出状态信号
    {
        for(temp=112;temp!=0;temp--)        //空循环
```



```
        {  
        }  
    }  
}
```

该程序中，通过双重 for 循环语句来执行空循环，以达到延时的目的。

28.2.2 终止信号

终止信号用于终止 I2C 总线通信。在时钟线 SCL 为高电平期间，数据线 SDA 上出现由低电平到高电平变化的上升沿时，被认为是终止信号。终止信号一出现，所有总线操作都结束，主从器件释放总线控制权。终止信号的时序，如图 28-3 所示。

如果采用汇编语言进行程序设计，则其程序示例如下：

```
STOP:    CLR        P1.1            ; P1.1=0 , SDA=0  
         SETB       P1.0            ; P1.0=1 , SCL=1  
         NOP  
         NOP  
         SETB       P1.1            ; P1.1=1 , SDA=1  
         NOP  
         NOP  
         CLR        P1.0            ; P1.0=0 , SCL=0  
         RET
```

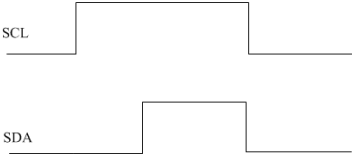


图 28-3 终止信号时序

该程序中，使用 P1.1 作为 SDA，P1.0 作为 SCL，通过 SETB 指令和 CLR 指令来实现终止信号的时序。NOP 指令用于延时，使其满足传输速度的要求。

如果采用 C51 语言进行程序设计，则其程序示例如下：

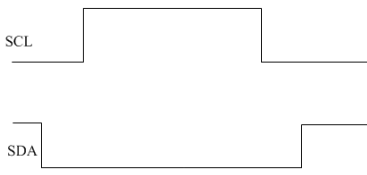
```
void Stop()                                //终止信号子程序  
{  
    SCL=0;  
    DelayMs(1);  
    SDA=0;  
    DelayMs(1);  
    SCL=1;  
    DelayMs(1);  
    SDA=1;  
    DelayMs(1);  
}
```

该程序中，通过直接为 SDA 和 SCL 赋值来实现终止信号的时序。其中的 DelayMs 函数为延时子程序，用于满足传输速度的要求。

28.2.3 应答信号

应答信号用于表明数据传输的结束。I2C 总线数据传输时，每传输一个字节数据后都必须有应答信号。应答信号从主器件产生。主器件在第 9 个时钟位上释放数据总线，使其处于高电平状态，此时从器件输出低电平拉低数据总线为应答信号。应答信号的时序，如图 28-4 所示。

如果采用汇编语言进行程序设计，则发送应答位程序示例如下：



```
ACK:    CLR        P1.1            ; P1.1=0 , SDA=0  
         SETB       P1.0            ; P1.0=1 , SCL=1  
         NOP  
         NOP  
         CLR        P1.0            ; P1.0=0 , SCL=0  
         NOP  
         NOP  
         SETB       P1.1            ; P1.1=1 , SDA=1
```

图 28-4 应答信号时序

RET

该程序中，使用 P1.1 作为 SDA，P1.0 作为 SCL，通过 SETB 指令和 CLR 指令来实现发送应答位的时序。NOP 指令用于延时，使其满足传输速度的要求。

如果采用 C51 语言进行程序设计，则发送应答位子程序示例如下：

```
void Ack() //发送应答位子程序
{
    SDA=0;
    DelayMs(1);
    SCL=1;
    DelayMs(1);
    SCL=0;
    DelayMs(1);
    SDA=1;
    DelayMs(1);
}
```

该程序中，通过直接为 SDA 和 SCL 赋值来实现发送应答位的时序。其中的 DelayMs 函数为延时子程序，用于满足传输速度的要求。

28.2.4 非应答信号

非应答信号用于数据传输出现异常而无法完成时。在传送完一个字节数据后，在第 9 个时钟位上从器件输出高电平为非应答信号。非应答信号的产生有两种情况。

- 当从器件正在进行其他处理而无法接收总线上的数据时，从器件不产生应答，此时从器件释放总线，将数据线置为高电平。这样，主器件可产生一个停止信号来终止总线数据传输。
- 当主器件接收来自从器件的数据时，接收到最后一个数据字节后，必须给从器件发送一个非应答信号，使从器件释放数据总线。这样，主器件才可以发送停止信号，从而终止数据传送。

非应答信号的时序，如图 28-5 所示。

如果采用汇编语言进行程序设计，则发送非应答位子程序示例如下：

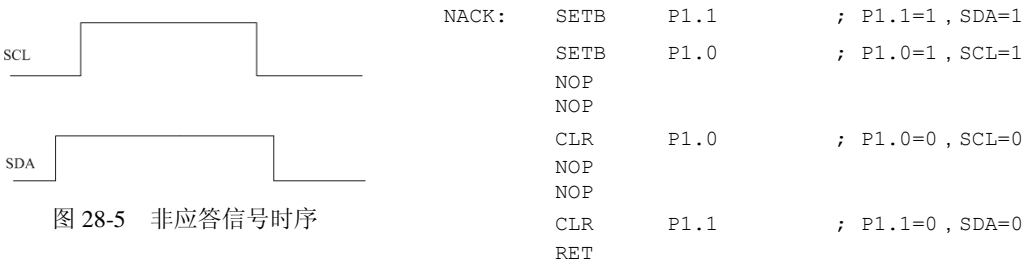


图 28-5 非应答信号时序

该程序中，使用 P1.1 作为 SDA，P1.0 作为 SCL，通过 SETB 指令和 CLR 指令来实现发送非应答位的时序。NOP 指令用于延时，使其满足传输速度的要求。

如果采用 C51 语言进行程序设计，则发送非应答位子程序示例如下：

```
void NoAck() //发送非应答位子程序
{
    SDA=1;
    DelayMs(1);
    SCL=1;
    DelayMs(1);
    SCL=0;
    DelayMs(1);
}
```

该程序中，通过直接为 SDA 和 SCL 赋值来实现发送非应答位的时序。其中的 DelayMs 函

数为延时子程序，用于满足传输速度的要求。

28.2.5 应答位检查

应答位检查用于检测接收的是否为正常的应答信号，以便于判断数据接收是否正常，方便后期处理。如果采用汇编语言进行程序设计，则检查应答位子程序示例如下：

```
CACK:   SETB     P1.1           ; P1.1=1, SDA=1
        NOP
        SETB     P1.0           ; P1.0=1, SCL=1
        NOP
        CLR      F0             ; 预设 F0=0, 表示正常应答信号
        MOV      A,    P1       ; 读端口 P1, 输入 P1.1/ SDA 引脚状态
        JNB      ACC.1,    CND   ; 检查 SDA 状态, 正常状态转向 CND
        SETB     F0             ; 无正常应答, F0=1, 表示非应答信号
CND:    CLR      P1.0           ; 结束子程序, 使 P1.0=0
        NOP
        RET
```

该程序中，采用 F0 作为应答检查的标志位，当检查到外围器件的正常应答后，置标志位 F0=0，表明被控制的器件接收到了主机发送的数据，否则置 F0=1。

如果采用 C51 语言进行程序设计，则检查应答位子程序示例如下：

```
bit TestAck() // 应答位检查子程序
{
    bit ErrorBit;
    SDA=1;
    DelayMs(1);
    SCL=1;
    DelayMs(1);
    ErrorBit=SDA; // 读入数据线上的应答状态
    DelayMs(1);
    SCL=0;
    DelayMs(1);
    return(ErrorBit); // 返回应答状态, 0 为正常应答信号, 1 为非应答信号
}
```

该程序中，定义了 ErrorBit 作为应答检查位，用于数据线 SDA 上的应答位检查结果，最后通过 return 语句返回该值。

28.2.6 总线数据位

在 I2C 总线启动后或应答信号后的第 1~8 个时钟脉冲，对应于要传送字节的 8 位数据，数据位由低到高传送。

I2C 总线上的数据是伴随着时钟脉冲，一位一位地传送的，每位数据占一个时钟脉冲。在时钟线 SCL 高电平期间，数据线 SDA 的状态就表示要传送的数据，高电平为数据 1，低电平为数据 0。在数据传送时，数据线上数据的改变在时钟线为低电平时完成，而时钟线为高电平时，数据线必须保持稳定，否则数据线上的任何变化都会被当做起始或终止信号，而致使数据传输停止。

28.2.7 写数据

I2C 总线协议规定了完整的数据传送格式。按照协议规定，数据传输的开始以主器件发出起始信号为准，然后发送寻址字节。寻址字节共 8 位，高 7 位是被寻址的从器件地址，最低一位是方向位，方向位表示主器件与从器件之间的数据传送方向，方向位为“0”时表示主器件向从器件发送数据（写）。在寻址字节后是将要传送的数据字节与应答位，数据可以多字节连

续发送。在数据传送完毕后，主器件必须发送终止信号已释放总线控制权。如果主器件希望继续占用总线，则可以不产生终止信号，马上再次发送起始信号，并对另一从器件进行寻址，便可进行新的数据传送。

主器件向从器件发送 n ($n \geq 1$) 个数据，数据传送方向在整个传送过程中不变，其数据传送格式如下，其中的 A 和 \overline{A} 为应答信号或非应答信号，是由从器件发送的，其余的均由主器件发送。

开始信号	从机地址	0	A	数据 1	A	数据 n	A/\overline{A}	停止信号
------	------	---	-----	------	-----	-------	--------	------------------	------

下面分别介绍如何用汇编语言和 C51 语言来进行 I2C 总线的写数据操作。

1. 写一个字节数据子程序

如果采用汇编语言进行程序设计，则写一个字节数据子程序示例如下：

```
WRBYT:  MOV      R0, #08H           ;将数据长度(8位)送入R0中
ZLL:    RLC      A                 ;发送数据左移,使发送的数据位进入C
        JC      W1                 ;判断,发送“1”转W1,发送“0”转W0
        AJMP     W0
ZLL1:   DJNZ     R0, ZLL           ;8位数据未发送完毕,转ZLL
        NOP
        RET                      ;结束子程序
W1:     SETB     P1.1              ;发送数据“1”
        SETB     P1.0
        NOP
        NOP
        CLR      P1.0
        CLR      P1.1
        AJMP     ZLL1             ;转向ZLL1
W0:     CLR      P1.1              ;发送数据“0”
        SETB     P1.0
        NOP
        NOP
        CLR      P1.0
        AJMP     ZLL1
```

该程序中，向外围器件发送一个字节的数。数据由低到高逐位传出，其中 W1 用于发送数据“1”，W0 用于发送数据“0”。调用本子程序前，需要将发送的数据送入寄存器 A 中。如果采用 C51 语言进行程序设计，则写一个字节数据子程序示例如下：

```
bit Write8Bit(unsigned char input)           //写一个字节数据子程序
{
    //input 为待发送的数据
    unsigned char temp;
    for(temp=8;temp!=0;temp--)                //循环移位,逐位发送数据
    {
        SDA=(bit)(input&0x80);                //取数据的最高位
        DelayMs(1);
        SCL=1;
        DelayMs(1);
        SCL=0;
        DelayMs(1);
        input=input<<1;                        //左移一位
    }
    return 1;
}
```

该程序中，input 为发送的数据，作为参数传递进入子程序中。子程序通过 for 循环语句，

逐位将数据发送到 I2C 数据总线上。该子程序的使用需要首先使用 Start 子程序启动总线传输。写一个字节数据子程序的流程图，如图 28-6 所示。

2. 写 n 个字节数据子程序

如果主器件需要发送 n 个字节的数据，则首先发送起始位，接着是寻址字节，然后是数据所要存入单元的首地址，外围器件此时产生正确的应答后，主器件便将开始 n 个字节的数据传输。如果采用汇编语言进行程序设计，则写 n 个字节数据子程序示例如下：

```
WRNB:    PUSH    PSW                ;现场保护
RLP:     MOV     PSW, #18H          ;选用工作寄存器 R3
        LCALL    START              ;调用子程序 START
        MOV     A,    #A1H          ;寻址字节 A1H 写入 A 中，用户可以根据需要改写
        LCALL    WRNB               ;调用 WRNB
        LCALL    CACK               ;调用 CACK，检查应答位
        JB      F0,    WRNB          ;如果是非应答位，则重发
        MOV     R1,    #SEND        ;主器件发送缓冲区的首地址#SEND 送入 R1
WR:       MOV     A,    @R1
        LCALL    WRNB
        LCALL    CACK
        JB      F0,    WRNB
        INC     R1
        DJNZ    NBYT,    WR          ;发送数据的个数
        LCALL    STOP
        POP     PSW
        RET
```

在本程序中，调用了 START、STOP、WRBYT、CACK 子程序。其中 NBYT 是待传送字节个数的存放单元，包括一个字节的器件子地址，即 n+1。#SEND 是主器件发送数据缓冲区的首地址，其中第一个字节为器件内部的子地址，然后才是发送的 n 个数据。调用本程序后，n 个字节数据依次传送到外围器件内的相应单元中。如果采用 C51 语言进行程序设计，则写 n 个字节数据子程序示例如下：

```
void WriteI2C(unsigned char *Wdata,unsigned char RomAddress,unsigned char number)
{
    Start();                        //写 n 个字节数据子程序
    Write8Bit(WriteDeviceAddress);  //启动
    TestAck();                      //写写器件的寻址地址
    Write8Bit(RomAddress);          //应答检查
    TestAck();                      //写入 I2C 器件内部的数据存储首地址
    for(;number!=0;number--)        //应答检查
    {
        Write8Bit(*Wdata);          //循环，逐个字节发送
        TestAck();                  //写一个字节
        Wdata++;                    //应答检查
    }
    Stop();                         //指针增加，指向下一个数据
    DelayMs(1);                    //停止
}
```

该程序中，调用了起始信号子函数 Start、终止信号子函数 Stop、应答检查子函数 TestAck 和发送单个字节数据的函数 Write8Bit。

WriteDeviceAddress 为写 I2C 外围器件的寻址字节，遵循寻址字节的格式。其声明如下：

```
#define WriteDeviceAddress 0xa0
```

该子程序有三个传入变量，Wdata 为待发送数据的首地址，RomAddress 为 I2C 外围器件的数据写入首地址，number 为写字节的个数。通过调用多次 Write8Bit 子程序，分别用于发送寻址字节、I2C 外围器件的内部写地址及数据。写 n 个字节数据子程序的流程图，如图 28-7 所示。

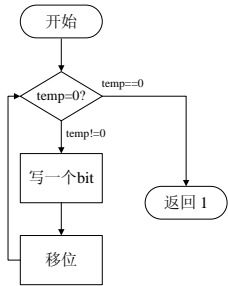


图 28-6 写一个字节数据子程序流程图

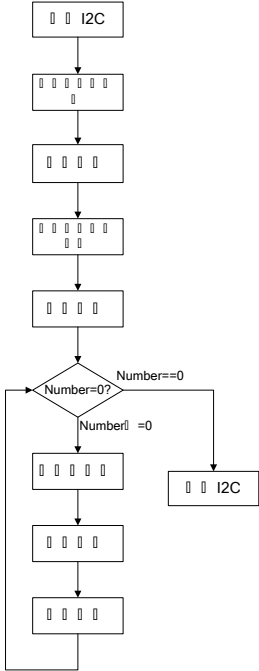


图 28-7 写 n 个字节数据子程序流程图

28.2.8 读数据

I2C 总线进行读数据时，数据传输的开始以主器件发出起始信号为准，然后发送寻址字节。寻址字节共 8 位，高 7 位是被寻址的从器件地址，最低一位是方向位，方向位表示主器件与从器件之间的数据传送方向，方向位为“1”时表示主器件从从器件中接收数据（读）。在寻址字节后是将要传送的数据字节与应答位，数据可以多字节连续发送。在数据传送完毕后，主器件必须发送终止信号已释放总线控制权。如果主器件希望继续占用总线，则可以不产生终止信号，马上再次发送起始信号，并对另一从器件进行寻址，便可进行新的数据传送。

主器件由从器件处读取 n (n≥1) 个数据，在整个传输过程中除寻址字节外，都是从器件发送、主器件接收，其数据传送格式如下，其中的最后一个非应答信号由主器件发送，其余应答信号和数据是从器件发送的。

开始信号	从机地址	1	A	数据 1	A	数据 n	\overline{A}	停止信号
------	------	---	---	------	---	-------	------	----------------	------

比较 I2C 总线数据读和数据写的传送格式，可以看出以下特点。

- 无论进行何种数据传输，寻址字节都是由主器件发出，数据字节的传送方向由寻址字节中方向位的定义而确定。
- 寻址字节只表明从器件地址及数据传送方向，从器件内部的 n 个数据地址，是由编程者在传送的第一个数据中指定的，即第一个数据为器件内存储单元的子地址，随后地址会自动加减，这样可以减少单元地址寻址操作。
- 每个字节传送完毕，后面都必须有一个应答信号 (A 或 \overline{A})。

下面分别介绍如何用汇编语言和 C51 语言来进行 I2C 总线的读数据操作。

1. 读一个字节数据子程序

如果采用汇编语言进行程序设计，则读一个字节数据子程序示例如下：

```
RDBYT:  MOV      R0, #08H          ;将数据长度(8位)送入寄存器R0中
RLP:    SETB     P1.1              ; P1.1=1, SDA=1
        SETB     P1.0              ; P1.0=1, SCL=1
        MOV      A, P1             ;读端口P1的引脚状态
        JNB      ACC.1, D0          ;判断, 读入“0”转向D0, 读入“1”转向D1
        AJMP     D1
RLP1:   DJNZ     R0, RLP            ;判断8位数据是否读完, 未完转向RLP
        RET
D0:     CLR      C                  ;读入“0”程序段
        MOV      A, R2
        RLC      A
        MOV      R2, A
        CLR      P1.0              ;置SCL=0, SDA可继续接收数据位
        AJMP     RLP1
D1:     SETB     C                  ;读入“1”程序段
        MOV      A, R2
        RLC      A
        MOV      R2, A
        CLR      P1.0              ;置SCL=0, SDA可继续接收数据位
        AJMP     RLP1
```

本子程序用来从外围器件上读取一个字节的数。执行本程序后，将逐位读取 8 位数据，并将其保存在 R2 中。其中 D0 用于处理读入“0”的程序段，D1 用于处理读入“1”的程序段。如果采用 C51 语言进行程序设计，则读一个字节数据子程序示例如下：

```
unsigned char Read8Bit()                //读一个字节数据子程序
{
    unsigned char temp,rbyte=0;
    for(temp=8;temp!=0;temp--)          //循环, 逐位读入字节数据
    {
        SCL=1;
        DelayMs(1);
        rbyte=rbyte<<1;                //左移一位
        DelayMs(1);
        rbyte=rbyte|((unsigned char)(SDA)); //数据线 SDA 上的数据存入 rbyte 的最低位
        SCL=0;
        DelayMs(1);
    }
    return(rbyte);                      //返回读入的字节数据
}
```

该子程序中，通过 for 语句来移位读取数据，其返回值为接收到的字节数据。该子程序的使用需要首先使用 Start 子程序启动总线传输。读一个字节数据子程序的流程图，如图 28-8 所示。

2. 读 n 个字节数据子程序

主器件读取 n 个字节数据的数据操作格式与发送 n 字节数据类似，只不过寻址地址中的方向位应置为读取。如果采用汇编语言进行程序设计，则读 n 个字节数据子程序示例如下：

```
RDNB:   PUSH     PSW
        MOV      PSW, #18H          ;选用工作寄存器 R3
```

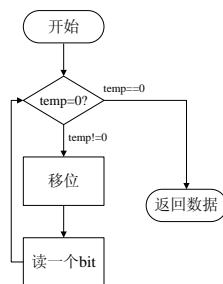


图 28-8 读一个字节数据子程序流程图

```

        LCALL    START          ;启动 I2C 总线，调用 START
        MOV      A,    #A1H     ;寻址字节 A1H 写入 A 中，用户可以根据需要改写
        LCALL    WRBYT          ;调用 WRBYT 发送寻址字节
        LCALL    CACK           ;检查应答位
        JB       F0,    RDNB     ;非正常应答则重新开始，否则继续执行
        MOV      A,    SUBA      ;器件内子地址送 A 中
        LCALL    WRBYT          ;发送子地址
        LCALL    CACK
        JB       F0,    RDNB     ;非正常应答则重新开始，否则继续执行
        LCALL    START          ;再一次启动总线
        MOV      A,    #A0H     ;寻址字节送入 A 中，用于读操作
        LCALL    WRBYT
        LCALL    CACK
        JB       F0,    RDNB     ;非正常应答位则重新开始
        MOV      R1,    #MBUFFER ;接收数据缓冲区首址 MBUFFER 送入寄存器 R1
READ1:   LCALL    RDBYT          ;读入一个字节到数据缓冲区中
        MOV      @R1, A
        DJNZ     NBYT, MORE     ;n 个字节是否读完，未完转入 MORE
        LCALL    NACK           ;读数据完毕，主器件发送非应答位
        LCALL    STOP           ;发送停止信号
        POP      PSW
        RET
MORE:    LCALL    ACK           ;发送应答位
        INC      R1             ;指向下一个接收数据缓冲区单元
        SJMP     READ1          ;转入读下一个字节数据
```

在本程序中，调用了 START、STOP、WRBYT、CACK、ACK、NACK 子程序。其中 MBUFFER 为主器件接收数据的缓冲区首地址，SUBA 为器件内部的子地址的存放单元。调用本程序后，n 个字节数据依次读入到主器件内的相应单元中。

需要注意的是，在读 n 个字节的操作中，除了发送寻址字节外，还要发送器件的子地址。因此，在读 n 个字节操作前，要进行一个字节的写操作，然后重新开始读操作，将从器件内的 n 个字节数据读出。如果采用 C51 语言进行程序设计，则读 n 个字节数据子程序示例如下：

```

void ReadI2C(unsigned char *RamAddress,unsigned char RomAddress,unsigned char bytes)
{
    Start();                      //读 n 个字节数据子程序
    Write8Bit(WriteDeviceAddress); //启动
    TestAck();                    //写写器件的寻址地址
    Write8Bit(RomAddress);        //应答检查
    TestAck();                    //写 I2C 器件内部数据的读取首地址
    Start();                      //应答检查
    Write8Bit(ReadDviceAddress);  //重新启动
    TestAck();                    //读写器件的寻址地址
    while(bytes!=1)               //应答检查
    {
        *RamAddress=Read8Bit();  //循环读入字节数据
        Ack();                  //读入一个字节
    }
}
```



```
RamAddress++; //地址指针递增
bytes--; //待读入数据个数递减
}
*RamAddress=Read8Bit(); //读入最后一个字节数据
NoAck(); //非应答
Stop(); //停止
}
```

该程序调用了起始信号子函数 **Start**、终止信号子函数 **Stop**、应答检查子函数 **TestAck**、发送单个字节数据的函数 **Write8Bit** 和读单个字节数据的函数 **Read8Bit**。

ReadDviceAddress 和 **WriteDeviceAddress** 分别为读写 I2C 外围器件的寻址字节，遵循寻址字节的格式。其声明如下：

```
#define WriteDeviceAddress 0xa0
#define ReadDviceAddress 0xa1
```

该子程序有三个传入变量，**RamAddress** 为接收数据缓冲区的首地址，**RomAddress** 为 I2C 外围器件的数据读取首地址，**bytes** 为读入字节的个数。该子程序调用多次 **Write8Bit** 子程序，分别用于发送读写寻址字节和 I2C 外围器件的内部数据读地址。其中循环调用 **Read8Bit** 子程序，用于逐个字节读取数据。读 **n** 个字节数据子程序的流程图，如图 28-9 所示。

28.3 51 单片机读写 EEPROM

I2C 总线接口器件以体积小、接口简单、读写操作方便等优点，使其在单片机系统中有着广泛的应用。目前常用于存储系统必要的参数，如密码、启动代码、设备标识等。例如，计算机主板中的 BIOS 就使用的是一个带有 I2C 总线的 EEPROM，其中保存了系统得重要信息和系统参数的设置程序。

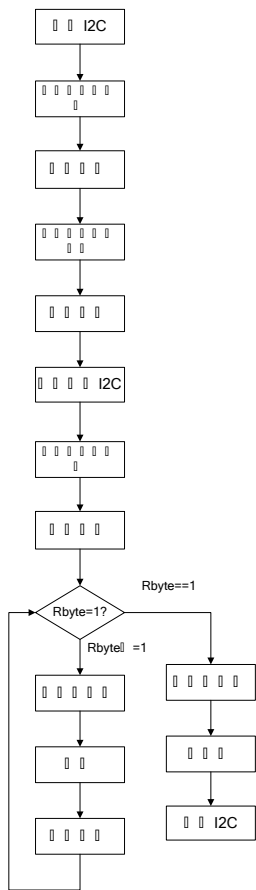
目前 USB 接口及其设备越来越被广泛使用，大有取代其他老式接口的趋势。然而，如何区分计算机上连接的众多 USB 外围设备呢？其实绝大部分的 USB 接口芯片都通过上电读一个带有 I2C 总线的串行 EEPROM，来载入该设备的 ID（包括 Vendor ID、Product ID 和 Device ID），根据这些 ID 来区分各个 USB 设备，并加载相应的驱动程序。

本节通过一个具体的实例，讲解如何利用 51 系列单片机通过 I2C 总线来烧写 USB 设备的 ID 标识。具体 USB 方面的介绍与设计可以参阅其他书籍。

目前最常用的是 Cypress 的 USB2.0 芯片 CY7C68013A，为了能让计算机区别不同的硬件设备，该芯片外接 I2C 总线接口的串行 EEPROM，其中保存了 USB 设备的标识符。计算机根据标识符来为该设备加载相应的驱动。这里选用 AT89S52 单片机来写入 USB 设备的标识符。本实例同样适用于其他型号的 24 系列串行 EEPROM 及 51 系列单片机。

28.3.1 串行 EEPROM 存储器简介

串行 EEPROM 存储器是一种采用串行总线的存储器，这类存储器具有体积小、功耗低、允许工作电压范围宽等特点。目前，单片机系统中使用较多的 EEPROM 芯片是 24 系列串行 EEPROM。其具有型号多、容量大、支持 I2C 总线协议、占用单片机 I/O 端口少，芯片扩展方便、读写简单等优点。



51 单片机开发与应用技术详解

目前，Atmel、MicroChip、National 等公司均提供各种型号的 I2C 总线接口的串行 EEPROM 存储器。下面以 Atmel 公司的产品为例进行介绍。

AT24C01/02/04/08 系列是 Atmel 公司典型的 I2C 串行总线的 EEPROM，这里以 AT24C08 为例介绍。AT24C08 具有 1024×8 位的存储容量，工作于从器件模式，可重复擦写 100 万次，数据可以掉电保存 100 年。8 引脚 DIP 封装的 AT24C08 的封装结构，如图 28-10 所示。

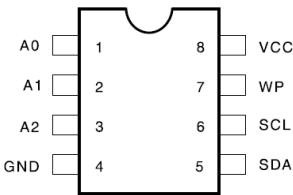


图 28-10 AT24C08 封装结构

- 其中，各个引脚定义如下：
- VCC、GND 为芯片的供电引脚；
 - A0~A2 为芯片的引脚地址和页面选择输入，而该芯片的器件地址为 1010；
 - SCL 为 I2C 总线的串行时钟线；
 - SDA 为 I2C 总线的串行数据线；
 - WP 为写保护引脚，当 WP 接高电平时，只能对该器件进行读操作，用于硬件数据的保护；当 WP 接低电平时，可以对该器件进行读写操作。

AT24C08 的所有操作都遵循 I2C 串行总线的操作时序。

28.3.2 电路设计

下面给出单片机 AT89S52 读写 AT24C08 的电路图，如图 28-11 所示。该电路图中所使用的元器件的参数及数量，如表 28-2 所示。

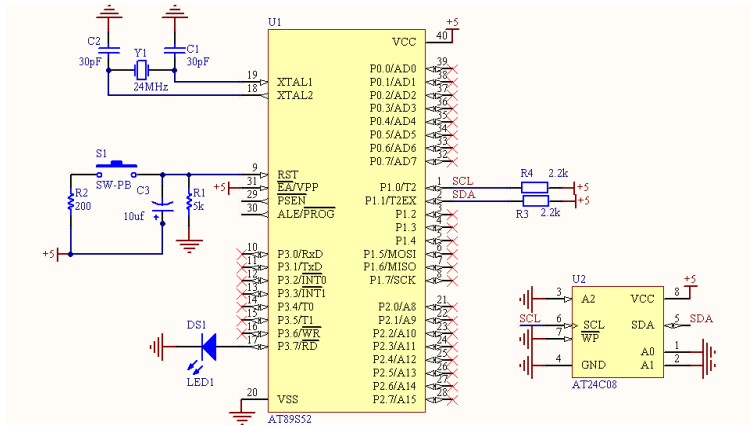


图 28-11 电路图

表 28-2 元器件列表

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89S52	1 个	R1	5k Ω	1 个
U2	AT24C08	1 个	R2	200 Ω	1 个
C1、C2	30pf	2 个	R3、R4	2.2k Ω 排阻	2 个
C3	10μf 电解	1 个	DS1	发光二极管	1 个
S1	按键开关	1 个	Y1	24MHz	1 个

其中，AT89S52 单片机只有最基本的复位系统和时钟电路，与 AT24C08 的接口只有两根线。AT24C08 的引脚 A0~A2 均接低电平，这样该芯片的地址为 1010000。因此，读器件的寻址字节为 10100001，即 A1，写器件的寻址字节为 10100000，即 A0。

I2C 总线的引脚分配如下：

- SCL：单片机的 P1.0 引脚，作为 I2C 串行总线的时钟线。
- SDA：单片机的 P1.1 引脚，作为 I2C 串行总线的数据线。

SCL 和 SDA 引脚均连接 2.2kΩ 的上拉电阻。

从引脚分配可以看出使用 I2C 串行总线和外围器件实现通信，具有连接简单、接口少等优点。对于通信速度要求不高而对体积要求较高的应用来说，采用 I2C 串行总线是一个不错的选择。

28.3.3 程序设计

1. 工程建立

这里采用 Keil C51 语言编写程序。具体操作步骤如下：

- (1) 首先在 μ Vision3 中，选择“Project”→“New”→“μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 28-12 所示。
- (3) 单击“确定”按钮，此时弹出“μ Vision3”对话框，如图 28-13 所示。单击“是”按钮，完成工程的建立。

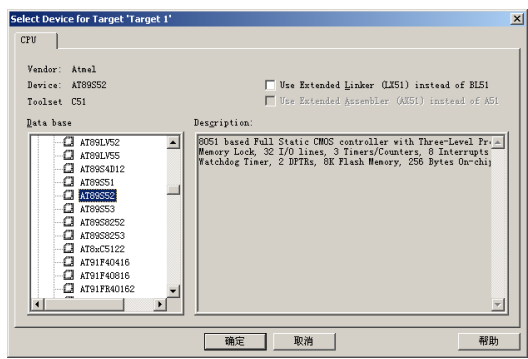


图 28-12 选择单片机 AT89S52

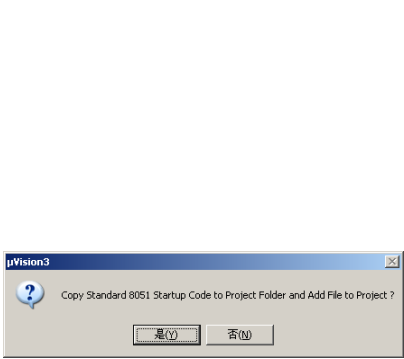


图 28-13 “μ Vision3”对话框

- (4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

2. 主程序

本例的程序功能是利用单片机的 P1.0、P1.1 作为 I2C 串行总线的 SCL、SDA 接口，向 AT24C08 写入 8 个字节的 USB 设备 ID 数据，然后再将写入的数据读出，并比较。如果读写的数据完全一致，则置 P3.7 引脚为高电平，发光二极管亮；否则置 P3.7 引脚为低电平，发光二极管灭。主程序的流程图，如图 28-14 所示。程序代码示例如下：

```
#include <reg51.h>

/*****
*****/

#define WriteDeviceAddress 0xa0 //写器件地址
#define ReadDviceAddress 0xa1 //读器件地址

/*****
*****/

sbit SCL=P1^0; //I2C 时钟线 SCL
sbit SDA=P1^1; //I2C 数据线 SDA
sbit DOG=P3^7; //程序运行标志及数
//据读写正确标志

/*****
*****/
```

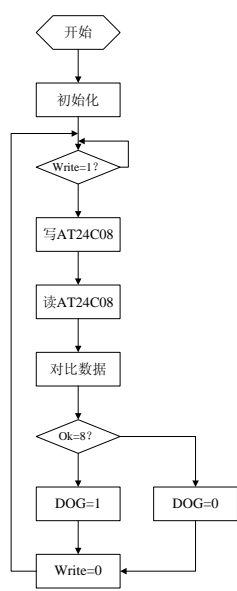


图 28-14 主程序流程图

51 单片机开发与应用技术详解

```
/*这部分是前面的各个 C 语言读写子程序，这里省略*/
/*****

void main()
{
    unsigned char writeByte[8]={0xC0,0x34,0x12,0x11,0x22,0x01,0x00,0x00};
                                //需要写的 8 个字节 USB 数据 ID

    unsigned char readByte[8];    //用于存读入的 8 个字节数据
    unsigned char *addw;          //写数据指针操作
    unsigned char *addr;         //读数据指针操作
    unsigned char i;
    unsigned char ok=0;

    bit write=1;                  //读写标志
    DOG=0;
    while(1)
    {
        if(write==1)             //当 write==1 时，执行写和读操作
        {
            addw=writeByte;       //写地址映射
            addr=readByte;        //读地址映射
            WriteI2C(addw,0x00,8); //写数据
            ReadI2C(addr,0x00,8);  //读数据
            for(i=0;i<8;i++)       //判断每个字节读写是否一致
            {
                if(writeByte[i]==readByte[i])
                {
                    ok++;
                }
            }
            if(ok==8)
            {
                DOG=1;             //当读写一致时，P3.7 输出高电平
            }
            else
            {
                DOG=0;             //当读写不一致时，P3.7 输出低电平
            }
            write=0;               //置 write==0，读写完毕
        }
    }
}
```

该程序在 while 语句中对 AT24C08 执行写操作，其中调用了 WriteI2C 函数和 ReadI2C 函数，这在前面已有介绍。如果数据写入完毕，则置 write==0，主程序便不再执行写入操作。对于写入的数据，其中首字节 0xC0 表示 USB 上电时从该 EEPROM 中读取 VID、PID 和 DID 的值。VID 为 0x1234，PID 为 0x2211，DID 为 0x0001，最后一个字节 0x00 为配置字节。

28.3.4 仿真分析

Keil μ Vision3 集成开发环境提供了很好的信号仿真功能，下面就利用其进行 I2C 串行总线的时序仿真分析。具体操作步骤如下：

- (1) 在 Keil μ Vision3 集成开发环境中，选择“Debug”→“Start/Stop Debug Session”命令，进入仿真分析模式。
- (2) 选择“View”→“Logic Analyzer Window”命令，打开信号分析窗口。
- (3) 选择“Debug”→“Logic Analyzer”命令，弹出“Logic Analyzer”对话框。在其中加

入需要分析的信号 SCL 和 SDA，如图 28-15 所示。

(4) 单击“Close”按钮，退出该对话框。

(5) 选择“Debug”→“Go”命令，开始执行仿真。此时信号分析窗口便出现了 SCL 和 SDA 上的仿真波形，如图 28-16 所示。

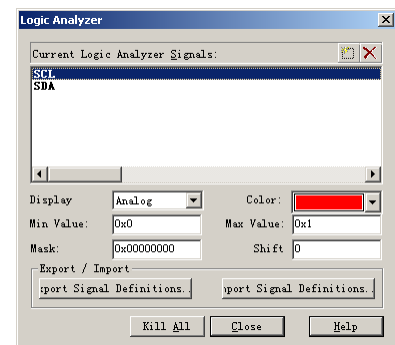


图 28-15 “Logic Analyzer”对话框

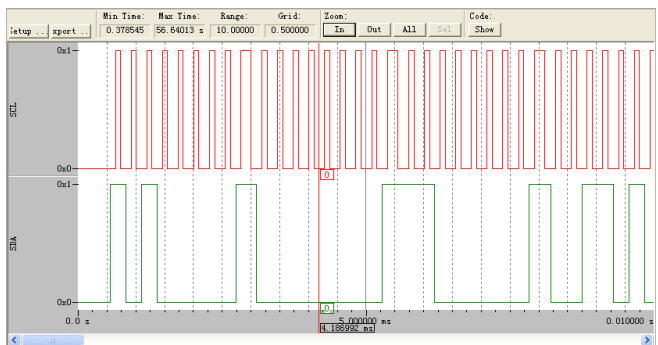


图 28-16 波形图

从仿真波形图中，可以分析单片机读写 AT24C08 的整个时序过程，包括起始信号、应答信号数据位及终止信号等。

28.4 小结

本章详细介绍了 I2C 串行总线的工作原理、结构及寻址方式等，并对 I2C 串行总线的数据传输进行了详细的介绍。本章还给出了采用普通的 51 系列单片机模拟读写 I2C 串行总线的汇编语言和 C 语言的代码。最后通过一个具体的实例，讲解了单片机读写 I2C 总线外围器件的电路设计及程序设计。I2C 串行总线具有接口简单，体积小等优点，在实际电路设计中经常使用。熟练掌握本章内容，可以控制大部分的 I2C 总线外围器件，从而大大扩展了 51 系列单片机的使用范围。

第 29 章 单片机音乐播放

单片机以其灵活的指令系统及强大的功能，除了在测控领域中有着广泛的应用外，还经常应用于智能玩具、电子贺卡等场合。在这些产品中，可以使用单片机驱动蜂鸣器发出声音，而且还可以控制其发出不同的声调，从而连接起来构成一首曲子。

目前市场上有很多种音乐芯片或音乐模块，可以直接产生各种曲子。但是这种模块价格比较贵，性价比不高。对于一些仅需要产生简单的音符或简短曲子的场合，可以使用单片机配合简单的蜂鸣器而产生需要的音乐效果，这种方法充分发挥了单片机的优势。

本章将主要介绍音调和节拍的基本原理，以及如何使用单片机进行音乐播放；然后将通过一个具体的实例来讲解如何使用单片机演奏曲子。

29.1 单片机发音概述

一般说来，单片机配合外接的蜂鸣器可以发出基本的单音频率。由于蜂鸣器发出的声音不包含相应幅度的谐波频率，因此采用这种方法不能演奏出多种音色的声音。使用单片机配合蜂鸣器来发音，只需弄清楚两个概念即可，也就是“音调”和“节拍”。其中，音调表示一个音符唱多高的频率；节拍表示一个音符唱多长的时间。下面分别介绍音调和节拍的相关知识，以及如何使用单片机来实现音调和节拍。

29.1.1 音调

音调是表示一个音符唱多高的频率，和平时所说的“音高”十分类似。这是音乐学中的名词，在音乐中常把中央 C 上方的 A 音定为标准音高，其频率 $f=440\text{Hz}$ 。其余音均通过和它比较获得。

如果 f_1 和 f_2 两个音符的频率相差一倍时，也即 $f_2=2\times f_1$ 时，则称 f_2 比 f_1 高一个倍频程。音符 1 (do) 与音符 $\dot{1}$ ，音符 2 (来) 与音符 $\dot{2}$ ，……，等之间正好相差一个倍频程，在音乐学中一般称其相差一个八度音。

音乐中规定，在一个八度音内，共有 12 个半音。以 1— $\dot{1}$ 八音区为例，这 12 个半音分别是 1— $\sharp 1$ 、 $\sharp 1$ —2、2— $\sharp 2$ 、 $\sharp 2$ —3、3—4、4— $\sharp 4$ 、 $\sharp 4$ —5、5— $\sharp 5$ 、 $\sharp 5$ —6、6— $\sharp 6$ 、 $\sharp 6$ —7、7— $\dot{1}$ 。由于人耳的听觉效果是非线性的，因此这 12 个音阶的分度基本上是以对数关系来划分的。

只要知道了这十二个音符的音调，也就知道了其基本音调的频率，这样可根据音符之间的倍频程关系得到各个音符基本音调的频率。知道了一个音符的频率后，便可以让单片机发出相应频率的振荡信号，从而驱动蜂鸣器产生相应的音符声音。

一般说来，常采用的方法就是通过单片机的定时器/计数器进行定时中断，在中断服务程序中将单片机上外接扬声器的 I/O 口来回置高电平或置低电平，从而驱动蜂鸣器发出声音。为了让单片机发出不同频率音符的声音，只需将定时器预置不同的定时值就可实现。

下面介绍如何确定一个频率所对应的定时器的定时值，这里以标准音高 A 为例，并假设单

片机外接 12MHz 的晶振，即 $f_0=12\text{MHz}$ 。标准音高 A 的频率 $f=440\text{Hz}$ ，其对应的周期如下：
 $T=1/f=1/440=2272\text{ }\mu\text{s}$
因此为了让蜂鸣器发出标准音高 A，则需要在单片机 I/O 端口输出周期为 $T=2272\text{ }\mu\text{s}$ 的方波脉冲，如图 29-1 所示。

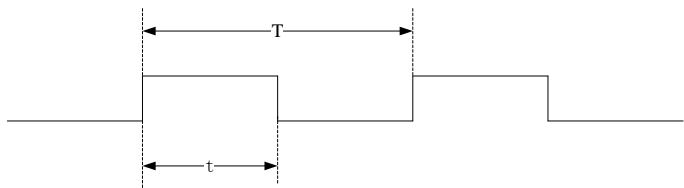


图 29-1 单片机控制音调示意图

这个方波脉冲的占空比为 1：1，因此单片机输出高电平和低电平信号均如下：
 $t=T/2=2272/2=1136\text{ }\mu\text{s}$
因此，在单片机中可以设置定时器的中断触发时间为 $1136\text{ }\mu\text{s}$ 。假定单片机定时器采用工作方式 1，以内部振荡器的 12 分频信号为计数脉冲，单片机外接晶振的振荡器频率为 f_0 ，则定时器的预置初值由下式来确定。
 $t=12\times(\text{Count}-\text{TT})\div f_0$
其中 TT 为定时器待确定的计数初值， $\text{Count}=2^{16}=65536$ 。定时器的计数初值 TH 和 TL 分别如下：

$\text{TH}=\text{TT}/256=(\text{Count}-t\times f_0/12)\div 256$
 $\text{TL}=\text{TT}\%256=(\text{Count}-t\times f_0/12)\%256$
这里将 $t=1136\text{ }\mu\text{s}$ 和 $f_0=12\text{MHz}$ 代入上面两式，其中要保证计算时时间和频率的单位换算一致。即可求出标准音高 A 在单片机定时器工作方式 1 下的定时器高低计数器的初值如下：
 $\text{TH}_A=(65536-1136\times 12/12)\div 256=\text{FBH}$
 $\text{TL}_A=(65536-1136\times 12/12)\%256=90\text{H}$
以上的求解方法同样适用于其他音调。下面给出部分音符的频率及单片机晶振频率 $f_0=12\text{MHz}$ ，定时器在工作方式 1 下的定时器高低计数器的预置初值，如表 29-1 所示。

表 29-1 音调对应的计数器预置值

C 调音符	频率/Hz	TH TL	C 调音符	频率/Hz	TH TL	C 调音符	频率/Hz	TH TL
1	262	F8 8B	1	523	FC 43	1̇	1045	FB 21
1 [#]	277	F8 F2	1 [#]	553	FC 78	1̇ [#]	1106	FE 3C
2	293	F9 5B	2	586	FC AB	2̇	1171	FE 55
2 [#]	311	F9 B7	2 [#]	621	FC DB	2̇ [#]	1241	FE 6D
3	329	FA 14	3	658	FD 08	3̇	1316	FE 84
4	349	FA 66	4	697	FD 33	4̇	1393	FE 99
4 [#]	370	FA B9	4 [#]	739	FD 5B	4̇ [#]	1476	FE AD
5	392	FB 03	5	783	FD 81	5̇	1563	FE C0
5 [#]	415	FB 4A	5 [#]	830	FD A5	5̇ [#]	1658	FE 02
6	440	FB 8F	6	879	FD C7	6̇	1755	FE E3
6 [#]	466	FB CF	6 [#]	931	FD E7	6̇ [#]	1860	FE F3

7	494	FC 0B	7	987	FE 05	7	1971	FF 02
---	-----	-------	---	-----	-------	---	------	-------

29.1.2 节拍

节拍表示一个音符唱多长的时间，其同样是音乐学中的名词。在一张完整乐谱的开头，都有如 $1=C \frac{4}{4}$ 、 $1=G \frac{3}{4}$ …… 等的标识。例如 $1=G \frac{3}{4}$ 的节拍示意图，如图 29-2 所示。

这里的 $\frac{4}{4}$ 、 $\frac{3}{4}$ 用来表示节拍，而 $1=C$ 、 $1=G$ 表示一个乐谱的曲调，简单地說就是跟音调有 关系。

$1=C \frac{3}{4}$

对于音符的节拍，这里以图 29-2 中所示的 $\frac{3}{4}$ 为例加以说 明。它表示乐谱中以四分音符为节拍，每一小节有三拍。在 图 29-2 中，1、2 为一拍，3、4、5 为一拍，6 为一拍。

图 29-2 节拍示意图

从发音的时长角度看，1、2 的时长为四分音符的一半， 即为八分音符长；3、4 的时长为八分音符的一半，即为十六 分音符长；5 的时长为四分音符的一半，即为八分音符长；6 的时长为四分音符长。

一般说来，如果乐曲没有特殊说明，则对于一拍的发音时间大约为 400~450ms。如果这 里规定一拍的时长为 400ms，那么以四分音符为节拍时，四分音符的时长为 400ms，八分音符 的时长为 200ms，十六分音符的时长则为 100ms。

在单片机上可以采用循环延时的方法来实现控制一个音符唱多长时间，从而实现节拍。一 般来说，首先需要编写一个精确地基本时长的延时程序，比如说以最短的十六分音符的时长为 基本延时时间。对于一个音符，如果它为十六分音符，则只需调用一次延时程序，如果它为四 分音符，则只需调用四次延时程序，如果它为二分音符，则只需调用八次延时程序，依次类推 即可。

29.1.3 单片机音乐播放的方法及音乐示例

前面介绍了基本音符的音调和节拍，并详细讲解了如何在单片机系统中来实现一个基本的 音符。在单片机上实现音乐播放，一般只需逐个播放音符即可。用单片机播发音乐的方法如下：

- (1) 初始化单片机定时器；
- (2) 将乐谱中的每个音符的音调及节拍变换成相应的音调参数和节拍参数；
- (3) 将乐谱中音符的参数做成数据表格，存放在存储器中；
- (4) 通过程序取出一个音符的相关参数，驱动蜂鸣器播放该音符；
- (5) 该音符唱完后，接着取出下一个音符的相关参数……，如此直到播放完毕最后一个音 符，根据需要也可循环不停地播放整个乐曲。

这里需要注意的是，对于一个乐曲中的休止符，一般将其音调参数设为 FFH，而其节拍参 数为 00H 来表示即可。

因为，一个音符需要音调和节拍两个参数来表示。这里将常用的音符所对应的计数器初值 放置在数组 music 中，以便于查表获得，示例如下：

```
unsigned char code music[30]=
{
    0xFF,0xFF,0xFB,0x90,0xFC,0x0C,0xFC,0x44,0xFC,0xAC,
    0xFD,0x09,0xFD,0x34,0xFD,0x82,0xFD,0xC8,0xFE,0x06,
    0xFE,0x22,0xFA,0x15,0xFB,0x04,0xFA,0x67,0xFE,0x85
};
```

对于一首音乐，可以将每个音符所代表的音调和频率信息组成一个数组。其中，每个元素

的前 4 位字节乘 2 表示为音符频率在数组 `music` 中的位置,后 4 位字节为多少个 1/4 拍。以 0x34 为例,其表示音符的音调所对应的计数器初值为 `music[6]` 和 `music[7]`,节拍数为 1 拍,共 4 个 1/4 拍。下面给出一些歌曲片断所对应的音符数组。

1. 《夜曲》

```
unsigned char code Mmusic[]=
{
    0xC2,0xB2,0x36,0x52,0x42,0x32,0x11,0xC1,0xB2,0xC6,
    0x22,0xC4,0x12,0xC2,0x76,0xA2,0x84,0x72,0x51,0x41,
    0x32,0x4C,0x41,0x36,0x74,0x82,0x74,0x82,0x72,0x52,
    0x41,0x38,0x32,0x42,0x14,0xC2,0x72,0x52,0x42,0x42,
    0x3C,0xC2,0xB2,0x36,0x52,0x42,0x32,0x11,0xC1,0xB2,
    0xC6,0x12,0xC4,0x12,0xC2,0x76,0xA2,0x82,0x72,0x51,
    0x41,0x32,0x4C,0x42,0x32,0x76,0x82,0x74,0x82,0x72,
    0x53,0x41,0x38,0x32,0x42,0x16,0xC2,0x72,0x52,0x41,
    0x53,0x12,0xCC,0xC2,0xB2,0x16,0xC2,0x72,0x52,0x42,
    0x41,0x3C,0x32,0x32,0x26,0x22,0x24,0x42,0x32,0x22,
    0x11,0x18,0x02,0x52,0x46,0x12,0x42,0x72,0x82,0x52,
    0x52,0x52,0x46,0x42,0x74,0x44,0x32,0x36,0x42,0x58,
    0x82,0x72,0x62,0x52,0x46,0x32,0x3C,0x14,0xB2,0x36,
    0x52,0x42,0x32,0x13,0xC1,0xB2,0xC6,0x12,0xC4,0x12,
    0xC2,0x76,0xA2,0x82,0x72,0x51,0x41,0x32,0x48,0x02,
    0x42,0x52,0x62,0x76,0x84,0x72,0x82,0x72,0x56,0x41,
    0x38,0x32,0x22,0x16,0xC2,0x72,0x52,0x43,0x43,0x3C,
    0x01,0x01,0x01,0x01,0xFF
};
```

2. 《祈祷》

```
unsigned char code Mmusic[37]=
{
    0xC4,0x14,0x32,0x44,0x44,0x52,0x52,0x44,0x32,0x32,
    0x14,0x14,0x12,0x12,0x32,0x42,0x54,0x54,0x4F,0xC4,
    0x12,0x32,0x44,0x42,0x72, 0x58,0x42,0x32,0x34,0x14,
    0x12,0x32,0x34,0x12,0x12,0x1F,0xFF
};
```

3. 《朋友》

```
unsigned char code Mmusic[97]=
{
    0x34,0x31,0x31,0x34,0x44,0x52,0x62,0x52,0x42,0x32,
    0x34,0x04, 0x74,0x74,0x62,0x62,0x64,0x3C,0x04, 0x64,
    0x62,0x52,0x42,0x32,0x34, 0x33,0x31,0x32,0x72,0x76,
    0x72,0x83,0x81,0x82,0x82,0x82,0x74,0x72,0x7C,0x04,
    0x63,0x61,0x64,0x64,0x64,0x72,0x82, 0x72,0x74,0x72,
    0x62,0x52,0x42,0x32, 0x42,0x44,0x42,0x42,0x52,0x62,
    0x52,0x5C,0x04, 0x64,0x62,0x62,0x64,0x72,0x82, 0x72,
    0x74,0x72,0x62,0x52,0x42,0x32, 0x42,0x46,0x53,0x41,
    0x42,0x32, 0x3C,0x04, 0x44,0x48,0x02,0x32, 0x3F, 0x44,
    0x48,0x01,0x34, 0x3F,0x44,0x0C, 0xFF
};
```

4. 《兰花草》

```
unsigned char code Mmusic[46]=
{
    0x22,0x52,0x52,0x56,0x56,0x42, 0x32,0x42,0x32,0x22,
    0x18, 0x82,0x82,0x82,0x82,0x86,0x72, 0xB2,0x72,0x72,
    0x62,0x58, 0x52,0x82,0x82,0x72,0x56,0x42, 0x32,0x42,
    0x32,0x22,0x16,0xB2, 0xB2,0x32,0x32,0x22,0x16,0x51,
    0x42,0x31,0x21,0xC1,0x88,0xFF
};
```

5. 《两只蝴蝶》

```
unsigned char code Mmusic[131]=
```

```
{
    0x52,0x44,0x52,0x02,0x42,0x42,0x42,0x38,0x04,0x12,
    0x30,0x44,0x42,0x52,0x42,0x32,0x12,0x12,0xC8,0x04,
    0x52,0x42, 0x58,0x02,0x42,0x52,0x42, 0x38,0x04,0x12,
    0x32, 0x44,0x42,0x52,0x42,0x32,0x12,0x32, 0x48,0x04,
    0x52,0x42, 0x58,0x02,0x42,0x52,0x42, 0x38,0x04,0x12,
    0x32, 0x44,0x42,0x52,0x42,0x32,0x12,0x11,0x31,0xC8,
    0x04,0x52,0x72, 0x78,0x01,0x72,0x82,0x72, 0x58,0x04,
    0x42,0x42,0x44,0x42,0x52,0x42,0x32,0x12,0x12,0x32,
    0x34,0x3C,0x09,0x72,0x72,0x82,0xA2,0x92,0x92,0x82,
    0x52,0x42,0x42,0x42,0x58,0x02,0x54,0x54,0x72,0x84,
    0x84,0x02,0x12,0x52,0x42,0x48,0x04,0x52,0x72,0x72,
    0x52,0x74,0x02,0xA4,0x92,0x82,0x92,0x54,0x02,0x82,
    0x82,0x92, 0x82,0x72,0x52,0x42,0x01,0xC4,0x11, 0x3C,
    0xFF
};
```

6. 《心心相印》

```
unsigned char code Mmusic[85]=
{
    0x24,0x14,0x24,0xC4,0x54,0x54,0x48,0x04,0x54,0x44,
    0x34,0x22,0x14,0xB2,0xC8,0x04,0x12,0xC2,0x16,0x12,
    0x76,0x52,0x46,0x12,0x3F,0x04,0xB4,0xD4,0xC4,0x14,
    0x28,0xC4,0x14,0x12,0x32,0x16,0xC2,0x54,0x78,0x34,
    0x4C,0x42,0x52,0x86,0x74,0x76,0x52,0x74,0x28,0x44,
    0x52,0x46,0x11,0x26,0x12,0xCF,0x04,0x24,0x14,0xC4,
    0x56,0x52,0x48,0x04,0x54,0x44,0x34,0x22,0x14,0xB2,
    0xC8,0x04,0x34,0x14,0xC4,0x12,0x32,0x18,0x42,0x42,
    0x74,0x52,0x34,0x3F,0xFF
};
```

7. 《最浪漫的事》

```
unsigned char code Mmusic[77]=
{
    0xC1,0xC1,0x34,0x42,0x42,0x52,0x42,0x51,0x5C,0x04,
    0xC2,0xC2,0x34,0x42,0x42,0x52,0x31,0x41,0x4C,0x04,
    0xC2,0xC2,0x34,0x42,0x42,0x52,0x41,0x54,0x58,0x32,
    0x54,0x4C,0x42,0x32,0x34,0x32,0x42,0x48,0x04,0xC2,
    0xC2,0x34,0x42,0x42,0x72,0x41,0x51,0x5C,0x04,0xC2,
    0xC2,0x32,0x32,0x42,0x42,0x52,0x42,0x4C,0x04,0xC2,
    0xC2,0x34,0x42,0x42,0x51,0x41,0x51,0x58,0x32,0x52,
    0x4C,0x42,0x32,0x32,0x42,0x4C,0xFF,0x54,0xC4,0x12,
    0x32,0x34,0x52,0x52,0x52,0x52,0x42,0x32,0x34,0xC1,
    0xC2,0x52,0xC2,0x11,0x32,0x34,0x11,0x12,0x31,0x12,
    0xC2,0x78,0x72,0x52,0x52,0x42,0x32,0x42,0x52,0x12,
    0xC1,0xC4,0xC1,0xC1,0xC1,0xC1,0x12,0x32,0x32,0x11,
    0x42,0x42,0x44,0xC2,0x12,0x32,0x42,0x52,0x72,0x72,
    0x82,0x42,0x51,0x42,0x48,0x42,0x12,0x3C,0x72,0x62,
    0x3C,0x72,0x62,0x3C,0x72,0x62,0x21,0x31,0x41,0x51,
    0x61,0x71,0x81,0x91,0x61,0x71,0x81,0x91,0x3F,0x3F,
    0x54,0xC2,0xC2,0x43,0xC2,0xC4,0x34,0x32,0x42,
    0x52,0xC2,0xC4,0xD2,0xC2,0x12,0x22,0x38,0x11,
    0x21,0x31,0x41,0x48,0xFF
};
```

8. 《一生有你》

```
unsigned char code Mmusic[138]=
{
    0xC1,0xC1,0x12,0x34,0x34,0x34,0x34,0x12,0x12,0x52,
    0x56,0x02,0xC2,0x12,0x32,0x32,0x32,0x34,0x52,
    0x72,0x7C,0x02,0x72,0x82,0x72,0x84,0x72,0x84,0x72,
    0x41,0x51,0x52,0x41,0x31,0x34,0x02,0x32,0x32,0x12,
    0x34,0x32,0x12,0x32,0x54,0x42,0x32,0x32,0x32,0x32,
    0x34,0x52,0x72,0x7C,0x72,0x72,0x82,0x72,0x82,0x74,
    0x84,0x82,0x72,0x42,0x52,0x42,0x34,0x12,0x12,0x32,
```

```

0xC4,0x52,0x42,0x34,0x12,0x12,0x31,0x41,0x32,0x3C,
0x0A,0x72,0x82,0xA2,0x84,0x82,0x72,0x82,0x72,0x52,
0x72,0x78,0x02,0x72,0x82,0xA2,0x84,0x82,0x72,0x82,
0x74,0x82,0x32,0x38,0x02,0x32,0x42,0x52,0x64,0x64,
0x72,0x82,0x82,0x72,0x82,0x82,0xA2,0x82,0x82,0x82,
0xA2,0x82,0x72,0x78,0x72,0x32,0x82,0x72,0x78,0x72,
0x82,0xA3,0x52,0x42,0x52,0x42,0x3C,0xFF
};

```

9. 《阳光总在风雨后》

```

unsigned char code Mmusic[144]=
{
    0x54,0x64,0x74,0x74,0x74,0x44,0x42,0x34,0x52,0x56,
    0x42,0x34,0x34,0x14,0x32,0x12,0x12,0xC2,0xC4,0x04,
    0x56,0x62,0x74,0x74,0x74,0x34,0x42,0x34,0x52,0x56,
    0x42,0x34,0x32,0x12,0x12,0x34,0x42,0x4C,0x52,0x42,
    0x44,0x32,0x42,0x32,0x12,0x12,0xC2,0xC2,0x54,0x52,
    0x54,0x52,0x52,0x42,0x32,0x32,0x12,0x32,0x42,0x32,
    0x52,0x5C,0x52,0x42,0x34,0x32,0x42,0x32,0x12,0x12,
    0xC2,0xC2,0x54,0x52,0x54,0x02,0x52,0x52,0x42,0x32,
    0x11,0x32,0x84,0x72,0x7F,0x0F,0x74,0x72,0x52,0x54,
    0x74,0x82,0x56,0x52,0x72,0x78,0x34,0x32,0x32,0x42,
    0x54,0x72,0x7F,0x86,0x82,0x72,0x52,0x54,0x72,0x84,
    0x32,0x36,0x52,0x42,0x52,0x42,0x32,0x34,0x12,0x32,
    0x32,0x44,0x4C,0x86,0x82,0x72,0x52,0x54,0x72,0x84,
    0x52,0x56,0x52, 0x41,0x52,0x72,0x82,0x52,0x34,0x11,
    0x11,0x31,0x3C, 0xFF
};

```

10. 《新水调歌头》

```

unsigned char code Mmusic[]=
{
    0x31,0x31,0x41,0x52,0x52,0x44,0x31,0x42,0x44,0x31,
    0x44,0x52,0x52,0x42,0x32,0x34,0x71,0x11,0x74,0x71,
    0x71,0x11,0x31,0x34,0x32,0x32,0x32,0x32,0x32,0x41,
    0x44,0x32,0x41,0x51,0x52,0x52,0x72,0x81,0x41,0x44,
    0x52,0x41,0x31,0x32,0x32,0x42,0x51,0x71,0x74,0x71,
    0x71,0x14,0x31,0x34,0x31,0x52,0x44,0x32,0x32,0x41,
    0x31,0x34,0x52,0x51,0x71,0x72,0x71,0x71,0x72,0x71,
    0x81,0x84,0x52,0x42,0x32,0x32,0x32,0x41,0x54,0x54,
    0x51,0x71,0x72,0x74,0x72,0x81,0xA1,0xA2,0x82,0x72,
    0x52,0x42,0x32,0x32,0x12,0x44,0x51,0x71,0x72,0x72,
    0x71,0x82,0x71,0x84,0x52,0x42,0x32,0x32,0x32,0x41,
    0x51,0x54,0x51,0x71,0x74,0x72,0x72,0x81,0xA1,0xA2,
    0x82,0x72,0x51,0x41,0x42,0x42,0x44,0x32,0x54,0x61,
    0x72,0x42,0x44,0x52,0x41,0x31,0x34,0x52,0x41,0x31,
    0x34,0x52,0x44,0x52,0x32,0x52,0x81,0x71,0x74,0x51,
    0x71,0x82,0x81,0x71,0x72,0x81,0x71,0x74,0x51,0x41,
    0x31,0x42,0x41,0x51,0x42,0x31,0x41,0x44,0x52,0x41,
    0x34,0x34,0x52,0x41,0x31,0x34,0x52,0x42,0x52,0x32,
    0x52,0x81,0x71,0x74,0x51,0x41,0x82,0x82,0xA4,0x71,
    0x81,0x72,0x71,0x51,0x41,0x31,0x44,0x41,0x31,0x41,
    0x51,0x44,0x52,0x52,0x42,0x31,0x34,0x52,0x42,0x32,
    0x42,0x32,0x33,0xFF
};

```

29.2 单片机音乐播放实例——电路图

前面介绍了单片机发声的原理，下面通过一个具体的实例来介绍如何使用单片机进行音乐演奏。系统中使用单片机外接扬声器来演奏一首音乐片断。整个系统的电路图如图 29-3 所示。电路中所使用的元器件，如表 29-2 所示。

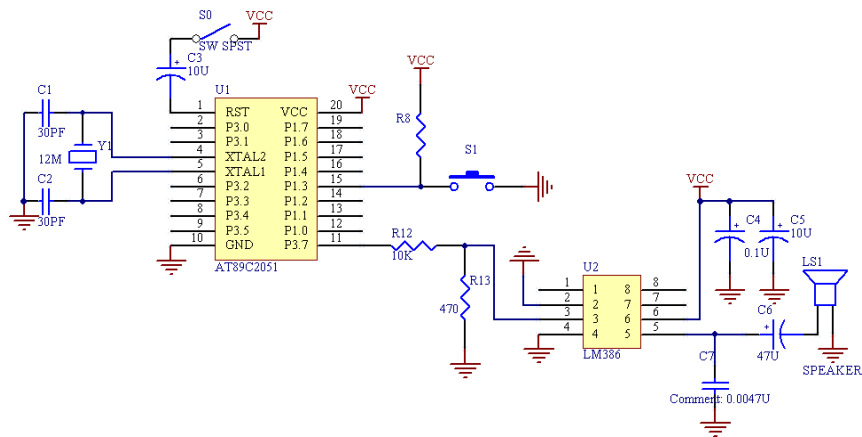


图 29-3 电路图

表 29-2 元器件

元 器 件	数 值	数 量
U1	AT89C2051	1 个
U2	LM386	1 个
Y1	12MHz	1 个
R8	2.2kΩ	1 个
R12	10kΩ	1 个
R13	470Ω	1 个
C1、C2	30pf	2 个
C3、C5	10μf 电解	2 个
C4	0.1 μf 电解	1 个
C6	47 μf 电解	1 个
C7	0.0047 μf 电解	1 个
LS1	扬声器	1 个
S0	电源开关	1 个
S1	播放按钮	1 个

这里单片机采用 AT89C2051，使用 LM386 作为功放芯片，扬声器采用普通小喇叭或蜂鸣器即可。电路中有一个电源总开关，为拨动开关。S1 为音乐播放按键，当 S1 按下时，开始演奏音乐。

29.3 单片机音乐播放实例——程序设计

这里采用 Keil C51 语言编写程序。

29.3.1 建立项目

首先在 Keil μ Vision3 集成开发环境中建立项目，具体操作步骤如下：

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89C2051，如图 29-4 所示。

(3) 单击“确定”按钮，此时弹出“μ Vision3”对话框，如图 29-5 所示。单击“是”按钮，完成工程的建立。

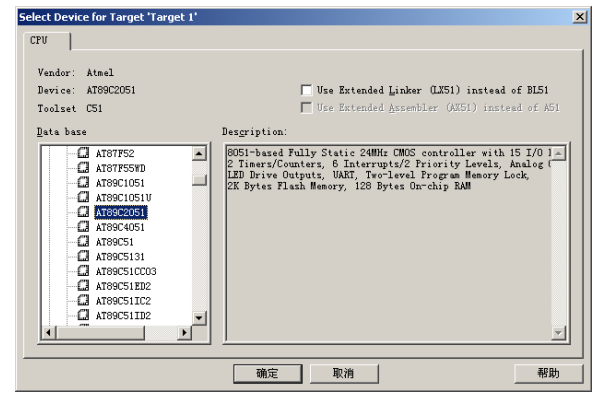


图 29-4 选择单片机 AT89C2051

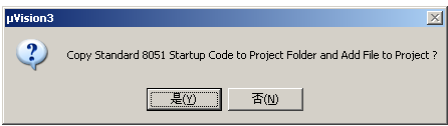


图 29-5 “μ Vision3”对话框

(4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

29.3.2 程序设计

主程序中判断播放按键是否按下，如果按下则开始播放音乐。程序代码示例如下：

```
#include <reg51.h>

sbit Y=P1^3; //播放按键

sbit Speaker=P3^7; //扬声器
static unsigned char bdata StateREG;
sbit m=StateREG^0;
unsigned char code * data Mymusic;
unsigned char data l;

void delay(void); //延时函数声明

unsigned char code music[30]=
{0xFF,0xFF,0xFB,0x90,0xFC,0x0C,0xFC,0x44,0xFC,0xAC,0xFD,0x09,0xFD,0x34,0xFD,0x82
,0xFD,0xC8,0xFE,0x06,0xFE,0x22,0xFA,0x15,0xFB,0x04,0xFA,0x67,0xFE,0x85};

unsigned char code Mmusic[]= //音乐数组
{
0x35,0x31,0x34,0x34,0x24,0x35,0x32,0x32,0x24,0x38,
0x44,0x58,0x48,0x34,0x31,0x34,0x34,0x24,0x38,0x34,
0x7F,0x32,0x34,0x32,0x34,0x24,0x38,0x34,0x24,0x38,
0x43,0x58,0x48,0x34,0x32,0x34,0x38,0x24,0x38,0x34,
0x7F,0x16,0x4C,0x74,0x78,0x64,0x54,0x48,0x54,0x64,
0x58,0x44,0x34,0x24,0x38,0x24,0x14,0x12,0x21,0x14,
0x78,0x68,0x3F,0x4C,0x74,0x78,0x64,0x52,0x42,0x48,
0x53,0x64,0x58,0x44,0x34,0x24,0x38,0x24,0x24,0x38,
0x44,0x58,0x48,0x3C,0xFF
};

void main() //主程序
{
    unsigned int data j;
    unsigned char data i;
    unsigned char data k;
```

51 单片机开发与应用技术详解

```

    unsigned char data p11;
    unsigned char data p33;

TMOD=0x01;                                //初始化
IE=0x82;

while(1)                                   //主循环
{
    j=0;
    m=0;

    while(Y==0)                            //当播放按键按下的时候
    {
        for(i=0;i<5;i++)
        {
            delay();                       //延时
        }
        p11=P1;
        p33=P3;
        while(*(Mmusic+j)!=0xFF)          //循环播放每个音符
        {
            k=*(Mmusic+j)&0x0F;
            l=*(Mmusic+j)>>4;
            if((p11!=P1)||((p33&0x0f)!= (P3&0x0f)))
            {
                goto Next;
            }

            TH0=music[2*l];                //查表获得计数器初值
            TL0=music[2*l+1];
            TR0=1;                          //启动定时器
            if ((music[2*l]==0xff)&&(music[2*l+1]==0xff))
            {
                TR0=0;                    //当播放结束的时候，停定时器
            }
            for(i=k;i>0;--i)
            {
                delay();
            }
            TR0=0;
            j++;
        }
        Next: ;                            //Next 语句，用于循环
    }
}
```

在主程序中，music[30]为音符所对应的计数器初值数组，Mmusic[]为音乐数组。主循环中使用 while 语句对按键进行判断，如果按键按下，则循环播放每个音符。其中使用移位语句分离出每个音符的音调和节拍，用于控制定时器播放音乐。

程序中使用定时器中断来控制扬声器来发声，程序示例如下：

```

void timer0() interrupt 1 using 1          //单个音符的发声
{
    TH0=music[2*l];
    TL0=music[2*l+1];
    Speaker = !Speaker;
}
```

另外，对于节拍的控制，使用了延时函数。这里采用汇编语句进行精确延时，延时 1/4 拍时间即 187ms。程序示例如下：

```

void delay(void)                          //延时
{
```

```

#pragma ASM                                //汇编嵌套
    MOV R3,#02H
D1: MOV R2,#7DH
D2: MOV R1,#0F8H
    DJNZ R1,$
    DJNZ R2,D2
    DJNZ R3,D1
#pragma ENDASM                            //汇编嵌套结束
}

```

程序编写完毕后，便可以进行编译。如果没有问题，则可以将程序下载到硬件中执行，当按下播放按键的时候，单片机便可以控制扬声器发出音乐。

29.4 小结

本章详细讲述了音乐学中音调和节拍的概念，以及如何使用单片机来实现音调和节拍的演奏。本章还给出了一些音乐片段的示例代码。最后通过一个具体的实例，讲解了如何使用单片机播放音乐。单片机的功能强大，成本低廉，对于需要简单音乐播放的场合，可以选择使用单片机控制扬声器来实现。

第 30 章 实时时钟芯片应用

对于一些测控系统或手持式设备，经常需要显示及设定时间。目前，市场上有多种实时时钟芯片提供了这类功能。这种可编程的实时时钟芯片内置了可编程的日历时钟及一定的 RAM 存储器，用于设定及保存时间。另外，实时时钟芯片一般内置闰年补偿系统，计时很准确。其采用备份电池供电，在系统断电时仍可以工作。实时时钟芯片的这些优点，使得其广泛应用于需要时间显示的场合。本章将详细介绍美国 DALLAS 公司推出实时时钟芯片 DS1302 的功能、控制指令及时间的操作。本章还将通过一个具体的实例，讲解如何使用 51 系列单片机来读写实时时钟芯片 DS1302。

30.1 实时时钟芯片 DS1302 概述

DS1302 是美国 DALLAS 公司推出的一款实时时钟芯片。其采用三线串行接口，芯片内部集成了可编程日历时钟和 31 个字节的静态 RAM。DS1302 的日历时钟可自动进行闰年补偿。DS1302 芯片自身还具有对备份电池进行涓流充电功能，可有效延长备份电池的使用寿命。

实时时钟芯片 DS1302 以其计时准确、接口简单、使用方便、工作电压范围宽和低功耗等优点，得到了广泛的应用。

30.1.1 实时时钟芯片 DS1302 概述

实时时钟芯片 DS1302 引脚排列，如图 30-1 所示。其中各个引脚功能如下所述。

- V_{cc1} (Pin1): 电源输入引脚，单电源供电时接 V_{cc1} 脚，双电源供电时用于接备份电源；
- V_{cc2} (Pin8): 电源输入引脚，双电源供电时用于接主电源；
- GND (Pin4): 接地引脚；
- X1, X2 (Pin2、Pin3): 时钟输入引脚，外接 32.768KHz 石英晶振；
- RST (Pin5): 复位引脚；
- I/O (Pin6): 数据输入/输出引脚；

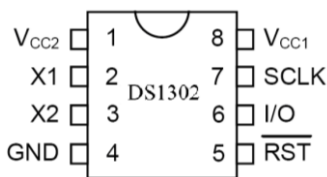


图 30-1 DS1302 的引脚排列

- SCLK (Pin7): 串行时钟输入引脚。

外部控制器可以通过 RST、SCLK 和 I/O 引脚来实现数据传送。其中，RST (Pin5) 为通信允许信号，低电平有效，即 RST=0 允许通信，RST=1 禁止通信。SCLK (Pin7) 为串行数据的位同步脉冲信号，I/O (Pin4) 为双向串行数据传送信号。

实时时钟芯片 DS1302 芯片的 X1 和 X2 引脚外接 32.768KHz 石英晶振，用于提供振荡源，供内部电路计时使用。

V_{cc1} 和 V_{cc2} 除了外接电源之外，还可以为备份电池充电。此时，可选用可充电镉镍电池或 $1\mu F$ 以上的超容量电容，内部涓流充电器在主电压工作正常时向备份电池充电，这样可以延长

电池的使用时间。这里需要注意的是，备份电池电压应略低于主电源的工作电压。

30.1.2 实时时钟芯片 DS1302 命令字节

实时时钟芯片 DS1302 为从器件，由外部微处理器来控制数据传输。每次传送时由 8051 向 DS1302 写入一个命令字节开始，后面是数据字节。实时时钟芯片 DS1302 命令字节的格式，如表 30-1 所示。

表 30-1 DS1302 写入命令字节的格式

位序	D7	D6	D5	D4	D3	D2	D1	D0
定义	1	RAM/CK	A4	A3	A2	A1	A0	RD/W

其中，命令字节各位的含义如下：

- 命令字的最高位 D7 为 1；
- RAM/CK 位为 DS1302 片内 RAM/时钟选择位，当 RAM/CK=1 时，为 RAM 操作，当 RAM/CK=0 时，为时钟操作；
- A4~A0 为片内日历时钟寄存器或 RAM 的地址选择位；
- RD/W 位为 DS1302 读写控制位。当 RD/W=1 时，为读操作，当 RD/W=0 时，为写操作。

实时时钟芯片 DS1302 执行读操作时，DS1302 接收命令字节后，按指定的选择对象及寄存器（或 RAM）地址，读取数据并通过 I/O 引脚传送给 8051 单片机；实时时钟芯片 DS1302 执行写操作时，DS1302 接收命令字节后，接收来自于 8051 单片机的数据并写入到 DS1302 相应的寄存器或 RAM 单元中。

30.1.3 实时时钟芯片 DS1302 数据格式

实时时钟芯片 DS1302 数据格式分为 RAM 和时钟两种操作，下面分别进行介绍。

1. 时钟操作

选择时钟操作时，与日期有关的 DS1302 数据格式，如图 30-2 所示。下面分别介绍各个寄存器含义。

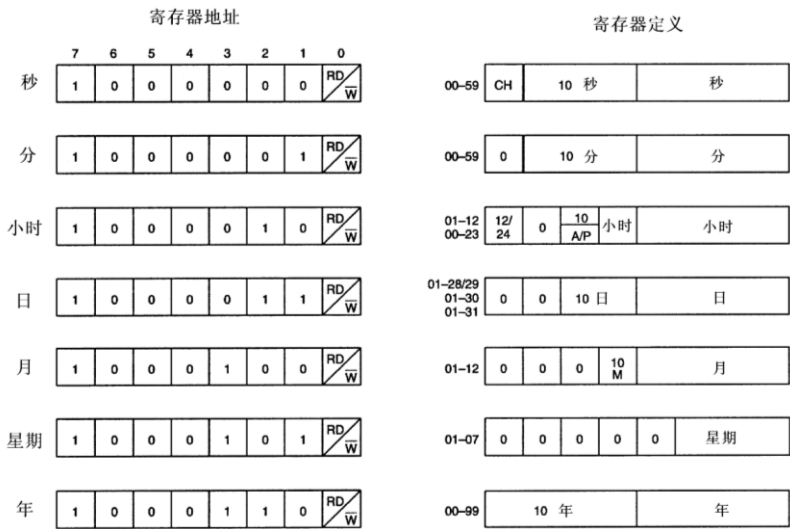


图 30-2 DS1302 的日期数据格式

- 秒寄存器，地址为 00H，其中以 BCD 码形式分别存放秒信息。秒寄存器的最高位为时

钟暂停控制位，该位为 0 时时钟振荡器暂停，DS1302 进入低功耗状态，该位为 1 时启动时钟。

- 分寄存器，地址为 01H，其中以 BCD 码形式分别存放分钟信息。
- 小时寄存器，地址为 02H，其中以 BCD 码形式分别存放小时信息。
- 日寄存器，地址为 03H，其中以 BCD 码形式分别存放日期信息。
- 月寄存器，地址为 04H，其中以 BCD 码形式分别存放月信息。
- 星期寄存器，地址为 05H，其中以 BCD 码形式分别存放星期信息。
- 年寄存器，地址为 06H，其中以 BCD 码形式分别存放年信息。

与控制有关的 DS1302 数据格式，如图 30-3 所示。下面分别介绍各个寄存器的含义。

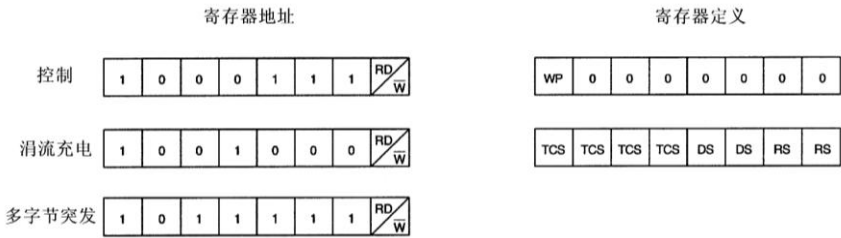


图 30-3 DS1302 的控制数据格式

- 控制寄存器，地址为 07H，用于写保护控制，只有当该寄存器的最高位 WP=0 时，才可以对日历时钟或 RAM 的内容进行写操作。
- 涓流充电控制寄存器，地址为 08H，控制内部涓流充电过程及充电电路的连接方式，该字节各位数据与涓流充电控制寄存器的关系，如图 30-4 所示。

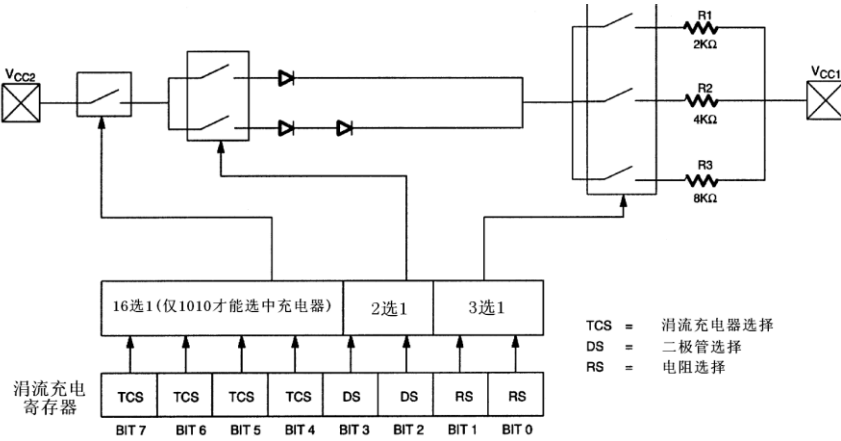


图 30-4 DS1302 的涓流充电控制寄存器

- 多字节突发方式 (burst) 控制寄存器，地址为 1FH，通过对该寄存器寻址，可以使用多字节方式对日历时钟或 RAM 进行读写操作。采用多字节突发方式写时钟寄存器时，必须按照数据传送的次序写入最先的 8 个寄存器；而以多字节方式写 RAM 操作时，为了传送数据不必写入全部的 31 个 RAM 字节。

2. RAM 操作

当选择片内 RAM 操作时，A4~A0 用于表示片内 RAM 单元地址，地址范围为 00H~1EH。DS1302 的 RAM 定义，如图 30-5 所示。

其中，地址 1FH 为 RAM 多字节命令。

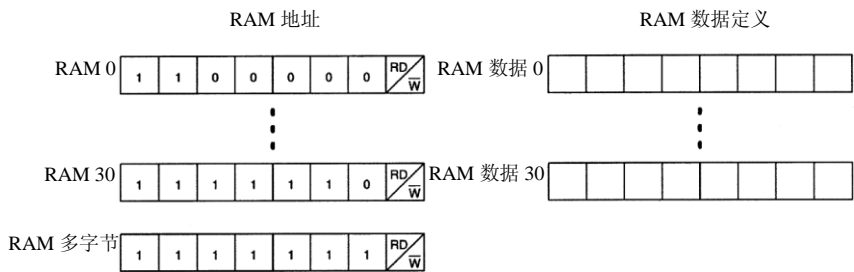


图 30-5 DS1302 的 RAM 定义

30.1.4 实时时钟芯片 DS1302 数据传输方式

实时时钟芯片 DS1302 与外部微处理器之间通过 I/O 引脚和 SCLK 引脚传送同步串行数据。其中，SCLK 为串行通信时的位同步时钟，一个 SCLK 脉冲传送一位数据。DS1302 在每次数据传送时以字节为单位，先发送低位，再发送高位，因此传送一个字节需要 8 个 SCLK 脉冲。

DS1302 的数据传输可采用单字节或多字节突发方式进行。下面分别进行介绍。

1. 单字节数据传输

DS1302 的数据以单字节方式传送时，RST=1，外部微处理器先向 DS1302 发送一个命令字节，紧接着发送一个字节的的数据，DS1302 在接收到命令字节后，自动将数据写入指定的片内地址或从该地址读取数据。单字节数据传输的时序，如图 30-6 所示。

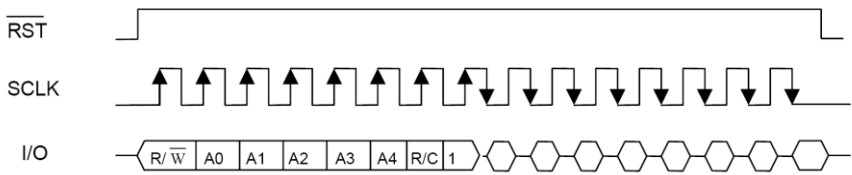


图 30-6 数据以单字节方式传送时序

2. 多字节数据传输

DS1302 的数据以多字节方式传送时，RST=1，外部微处理器先向 DS1302 发送的命令字节中 A4~A0 全为 1，则 DS1302 在接收到命令字节后，可以一次完成 8 个字节日历时钟寄存器数据或是片内 31 个字节 RAM 单元数据的读写操作。多字节数据传输的时序，时序如图 30-7 所示。

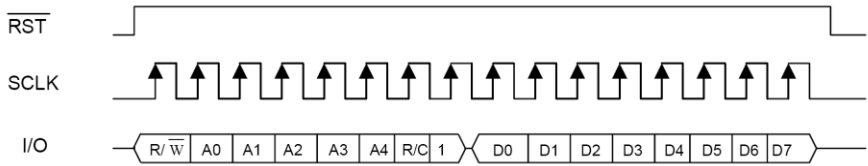


图 30-7 数据以多字节方式传送时序

从上面的介绍可知，DS1302 的单字节方式传送一次数据需要 16 个 SCLK 脉冲，如果写满 8 个日历时钟寄存器则需要 16×8=128 个 SCLK 脉冲，如果写满片内 31 个字节的 RAM 则需要 16×31=496 个 SCLK 脉冲。多字节方式完成对 8 个日历时钟寄存器读写时需要 72 个 SCLK 脉冲，而对片内 RAM 单元读写时则最多需要 256 个 SCLK 脉冲。

这两种方式各有优势，单字节数据传输方式可保证数据传送时的安全性和可靠性，多字节

数据传输方式则可提高数据传送速度。在使用时，可根据需要灵活选用。

30.2 单片机读写实时时钟芯片实例

实时时钟芯片 DS1302 采用串行接口，只需三根引脚便可以完成操作，使用非常方便。这里采用 AT89S52 单片机作为外部微处理器，来对 DS1302 进行控制。当然也可以选择其他兼容的 8051 单片机。

30.2.1 电路图

下面给出单片机 AT89S52 读写实时时钟芯片 DS1302 的电路图，如图 30-8 所示。该电路图中所使用的元器件的参数及数量，如表 30-2 所示。

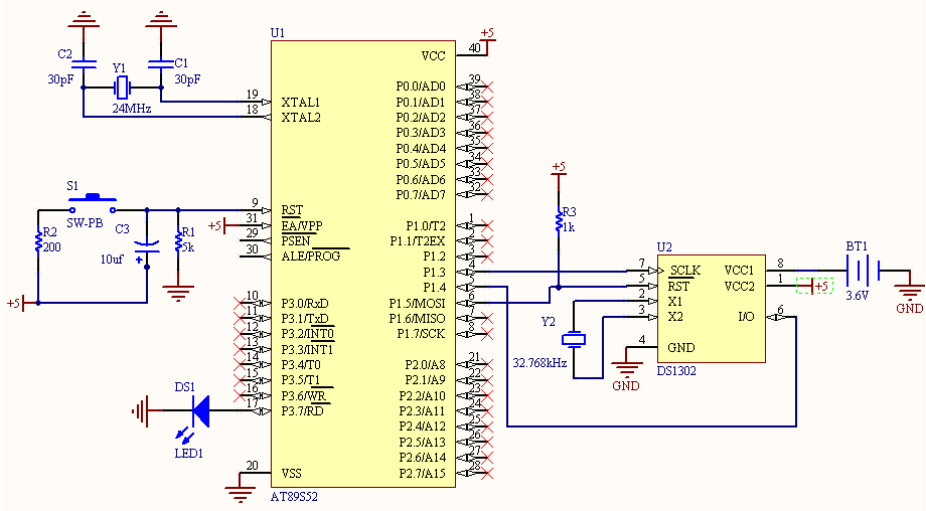


图 30-8 电路图

表 30-2 元器件列表

元 器 件	数 值	数 量
U1	AT89S52	1 个
U2	DS1302	1 个
C1、C2	30pf	2 个
C3	10μf 电解	1 个
S1	按键开关	1 个
R1	5k Ω	1 个
R2	200 Ω	1 个
R3	1 k Ω	1 个
R3、R4	2.2k Ω 排阻	2 个
DS1	发光二极管	1 个
Y1	24MHz	1 个
Y2	32.768kHz	1 个
BT1	3.6V 备份电池	1 个

30.2.2 建立项目

这里采用 Keil C51 语言编写 AT89S52 上的程序,来实现对 DS1302 的控制。首先在 Keil μ Vision3 集成开发环境中建立项目, 具体操作步骤如下:

- (1) 在 μ Vision3 中, 选择“Project”→“New”→“μ Vision Project”命令, 新建一个工程, 并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52, 如图 30-9 所示。
- (3) 单击“确定”按钮, 此时弹出“μ Vision3”对话框, 如图 30-10 所示。单击“是”按钮, 完成工程的建立。

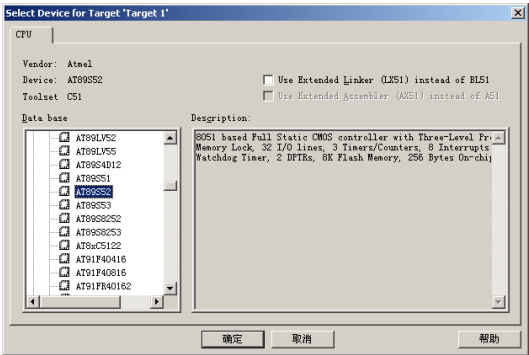


图 30-9 选择单片机 AT89S52



图 30-10 “μ Vision3”对话框

- (4) 选择“File”→“New”命令, 新建一个程序文件, 并保存为*.C 文件, 可以在其中输入程序代码。

30.2.3 主程序

本例的程序功能是首先初始化 DS1302, 然后通过输入不同的字符来分别执行初始化、时钟操作和 RAM 操作。分别介绍如下:

- 输入字符 c 或 C, 进入时钟操作。此时可以选择显示时间(输入字符 d 或 D)或写时钟(输入字符 w 或 W);
- 输入字符 i 或 I, 进入初始化操作。此时可以详细设置时间和日期等;
- 输入字符 r 或 R, 进入 RAM 操作。此时可以选择单字节写 RAM(输入字符 b 或 B)、多字节写 RAM(输入字符 w 或 W)或多字节读 RAM(输入字符 r 或 R)。

主程序的流程图, 如图 30-11 所示。程序代码示例如下:

```
#include <reg52.h> //头文件
#include <stdio.h>

#define uchar unsigned char //宏定义

sbit SCLK = P1^3; //声明接口引脚
sbit I_O = P1^4;
sbit RSTB = P1^5;

void ResetDS1302(); //复位函数声明
void InitDS1302(); // DS1302 初始化函数声明
void WriteClkByte(); //时钟字节写入函数声明
void WriteRamByte(); // RAM 字节写入函数声明
```

51 单片机开发与应用技术详解

```
uchar    ReadByteDS1302();           //字节读取函数声明
void     WriteByteDS1302(uchar);     //字节写入函数声明
void     Display(uchar);             //显示时钟寄存器内容函数声明
void     BurstReadRAM();             //多字节突发方式读取 RAM 函数声明
void     BurstWriteRAM();            //多字节突发方式写入 RAM 函数声明

void main (void)                     //主函数
{
    uchar Key, Key1,Key2;
    SCON  = 0x50;                     //设置串行口：方式 1，8 位 UART，允许接收
    TMOD  |= 0x20;                     //设置定时器 T1，方式 2，8 位自动重装
    PCON  =0x80;                       //设置 SMOD=1
    TL1   =0xF4;                       //波特率 4800bit/s，初值
    TH1   =0xF4;
    TR1   = 1;                         //开启定时器 1
    TI    = 1;

    printf("*****AT89S52 CONTROL DS1302*****\n\n");
    printf("Initialize the DS1302.\n");
    InitDS1302();                      //初始化 DS1302
    while (1)
    {
        printf("\nEnter DS1302 Menu Selection:");
        printf("c or C: Run the Clock Routine\n");
        printf("i or I: Initialize the DS1302\n");
        printf("r or R: Run the RAM Routine\n");

        Key = _getkey();                //获取输入字符
        switch(Key)
        {
            case 'c':                    //执行时钟操作
            case 'C':
                printf("\rRun the Clock Routine\n");

                Key1 = _getkey();         //获取输入字符
                switch(Key1)
                {
                    case 'd':             //显示
                    case 'D':
                        Display(1);
                        break;

                    case 'w':             //时钟字节写入
                    case 'W':
                        WriteClkByte();
                        break;
                }
                break;

            case 'i':                     //初始化 DS1302
            case 'I':
                InitDS1302();
                break;

            case 'r':                     //RAM 操作
            case 'R':
                printf("\rRun the RAM Routine\n");
```

```
Key2 = _getkey();           //获取输入字符
switch(Key2)
{
    case 'b':                //RAM 字节写入
    case 'B':
        WriteRamByte();
        break;

    case 'r':                //多字节突发方式读取 RAM
    case 'R':
        BurstReadRAM();
        break;

    case 'w':                //多字节突发方式写入 RAM
    case 'W':
        BurstWriteRAM();
        break;
}
break;
}
}
```

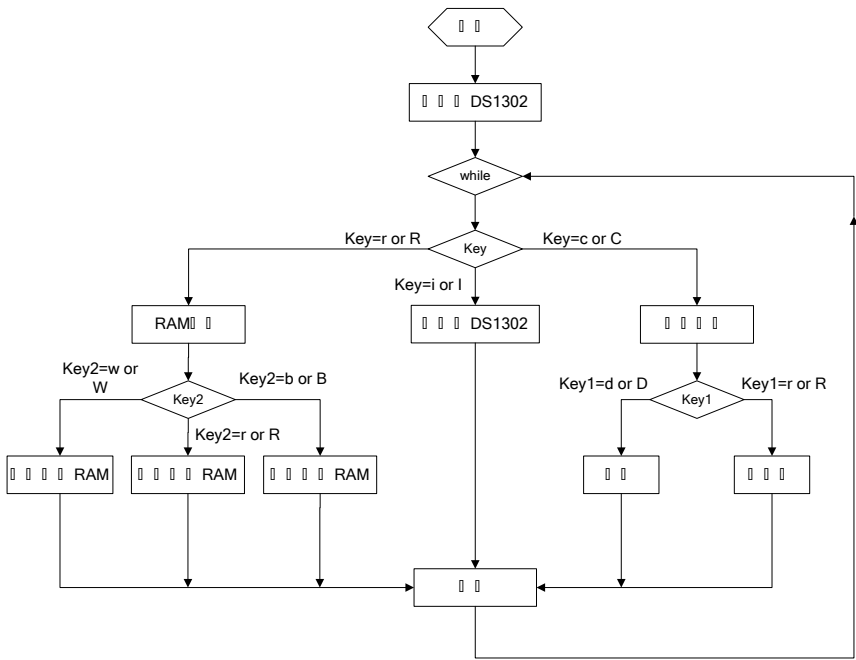


图 30-11 主程序流程图

在该程序中，首先定义了 DS1302 和 AT89S52 的接口引脚。在主函数中，首先初始化串口为模式 1，并设置定时器 T1 为 8 位自动重载方式，波特率设置为 4800bit/s，随后启动定时器 T1。接着，调用 InitDS1302 函数初始化 DS1302。在 while 主循环中，通过 _getkey 函数获取输入字符，来分别执行时钟操作、RAM 操作及复位 DS1302 等。在这里分别演示了时钟寄存器的读写操作及 RAM 的单字节写和多字节突发读写操作。

在主程序中，分别调用各个子函数来实现对 DS1302 的控制操作。下面分别进行介绍。

30.2.4 复位函数

复位函数 `ResetDS1302` 实现对实时时钟芯片 `DS1302` 的硬件复位操作。程序中使用 `RSTB` 引脚来进行复位操作，其程序代码示例如下：

```
void ResetDS1302()
{
    SCLK = 0;
    RSTB = 0;
    RSTB = 1;
}
```

30.2.5 字节读取函数

字节读取函数 `ReadByteDS1302` 用于读取 `DS1302` 的内部数据。该函数中，在 `SCLK` 引脚时钟下，通过 `I/O` 引脚逐位读取 8 位的字节数据，并通过 `return` 语句返回。其程序代码示例如下：

```
uchar ReadByteDS1302()
{
    uchar i;
    uchar RByte;
    uchar TempByte;

    RByte = 0x00;
    I_O = 1;
    for(i=0; i<8; ++i)                                //逐位读取字节数据
    {
        SCLK = 1;                                       //时钟操作
        SCLK = 0;
        TempByte = (uchar)I_O;
        TempByte <<= 7;                                  //移位
        RByte >>= 1;
        RByte |= TempByte;
    }
    return RByte;                                       //返回结果
}
```

30.2.6 字节写入函数

字节写入函数 `WriteByteDS1302` 用于向 `DS1302` 写入寄存器地址或数据。该函数中，在 `SCLK` 引脚时钟下，通过 `I/O` 引脚逐位写入 8 位的字节数据。其程序代码示例如下：

```
void WriteByteDS1302(uchar W_Byte)
{
    uchar i;
    for(i = 0; i < 8; ++i)                              //循环逐位写入
    {
        I_O = 0;
        if(W_Byte & 0x01) I_O = 1;
        SCLK = 0;                                       //时钟操作
        SCLK = 1;
        W_Byte >>= 1;                                    //移位
    }
}
```

30.2.7 初始化函数

初始化函数 `InitDS1302` 用于对 `DS1302` 的时钟寄存器进行初始化操作。该函数中，分别输入需要设置的年、月、星期、日、小时、分钟和秒的数值，然后调用 `ResetDS1302` 函数复位 `DS1302`，并允许涓流充电，最后以多字节突发方式写入时钟数据。其程序代码示例如下：


```

void InitDS1302()
{
    uchar    year, month, date, day, hour, minute, second;

    printf("\nEnter the Clock information: ");
    printf("\nPlease Enter the year (0-99): ");    //输入年
    scanf("%bx", &year);
    printf("\n Please Enter the month (1-12): ");    //输入月
    scanf("%bx", &month);
    printf("\n Please Enter the date (1-31): ");    //输入日
    scanf("%bx", &date);
    printf("\n Please Enter the day (1-7): ");    //输入星期
    scanf("%bx", &day);
    printf("\n Please Enter the hour (1-24): ");    //输入小时
    scanf("%bx", &hour);
    hour =hour & 0x3f;    //设置时钟为 24 小时方式
    printf("\n Please Enter the minute (0-59): ");    //输入分钟
    scanf("%bx", &minute);
    printf("\n Please Enter the second (0-59): ");    //输入秒
    scanf("%bx", &second);

    ResetDS1302();    //复位 DS1302
    WriteByteDS1302(0x8e);    //写保护控制寄存器
    WriteByteDS1302(0);    //允许写入

    ResetDS1302();    //复位 DS1302
    WriteByteDS1302(0x90);    //涓流充电寄存器
    WriteByteDS1302(0xab);    //允许充电

    ResetDS1302();    //复位 DS1302
    WriteByteDS1302(0xbe);    //以多字节突发方式写入 8 个字节时
                                //钟数据
    WriteByteDS1302(second);    //写入秒
    WriteByteDS1302(minute);    //写入分钟
    WriteByteDS1302(hour);    //写入小时
    WriteByteDS1302(date);    //写入星期
    WriteByteDS1302(month);    //写入月
    WriteByteDS1302(day);    //写入日
    WriteByteDS1302(year);    //写入年
    WriteByteDS1302(0);    //对写保护控制寄存器写入 0
    ResetDS1302();    //复位 DS1302
}

```

30.2.8 时钟字节写入函数

时钟字节写入函数 **WriteClkByte** 用于对时钟寄存器执行写操作。该函数中，首先获取时钟寄存器地址及时钟数据，然后分别调用 **WriteByteDS1302** 函数来执行写寄存器地址和写数据操作。其程序代码示例如下：

```

void WriteClkByte()
{
    uchar Address;

```

51 单片机开发与应用技术详解

```
uchar Data;

printf("\nWrite Clock Function:");
printf("\nWrite Clock ADDRESS:");
scanf("%bu", &Address); //获取地址
printf("\nWrite Clock DATA:");
scanf("%bx", &Data); //获取数据

Address = ((Address * 2) | 0x80); //时钟数据写入命令
ResetDS1302(); //复位 DS1302
WriteByteDS1302(Address); //写地址
WriteByteDS1302(Data); //写数据
ResetDS1302(); //复位 DS1302
}
```

30.2.9 RAM 字节写入函数

RAM 字节写入函数 WriteRamByte 用于对 DS1302 的片内 RAM 执行写操作。该函数中，首先获取 RAM 地址及时钟 RAM 数据，然后分别调用 WriteByteDS1302 函数来执行写寄存器地址和写数据操作。其程序代码示例如下：

```
void WriteRamByte()
{
uchar Address;
uchar Data;

printf("\nWrite RAM Function:");
printf("\nWrite Ram ADDRESS (HEX):");
scanf("%bx", &Address); //获取 RAM 地址
printf("\nWrite Ram DATA (HEX):");
scanf("%bx", &Data); //获取 RAM 数据

Address = ((Address * 2) | 0xC0); //RAM 数据写入命令
ResetDS1302(); //复位 DS1302
WriteByteDS1302(Address); //写 RAM 地址
WriteByteDS1302(Data); //写 RAM 数据
ResetDS1302(); //复位 DS1302
}
```

30.2.10 时钟寄存器内容显示函数

时钟寄存器内容显示函数 Display 用于读取并显示 DS1302 内部时钟寄存器中的数据。该函数中，首先复位 DS1302，然后采用多字节突发方式读取时钟寄存器中的数据，然后每秒显示一次结果。其程序代码示例如下：

```
void Display(uchar loop)
{
uchar lsec = 99, sec, min, hrs, dte, mon, day, yr;

do //主循环
{
ResetDS1302();
WriteByteDS1302(0xBF); //以多字节方式读取时钟寄存器数据
sec = ReadByteDS1302(); //读取秒
min = ReadByteDS1302(); //读取分
}
```

```

hrs = ReadByteDS1302();           //读取小时
dte = ReadByteDS1302();           //读取日期
mon = ReadByteDS1302();           //读取月份
day = ReadByteDS1302();           //读取星期
yr = ReadByteDS1302();            //读取年
ResetDS1302();                    //复位 DS1302
if(sec != lsec || !loop)           //每秒显示一次
{                                  //输出结果
    printf("\nYr  Day Mon  Dte Hrs Min Sec");
    printf("\n%2.bX  %2.bX  %2.bX  %2.bX", yr, day, mon, dte);
    printf("  %2.bX  %2.bX  %2.bX", hrs, min, sec);
    lsec = sec;
}
if(!loop) break;
}while(!RI);
if(loop) _getkey();
}

```

30.2.11 多字节突发方式读取 RAM 函数

多字节突发方式读取 RAM 函数 `BurstReadRAM` 用于读取 DS1302 内部 RAM 的数据。该函数中, 首先以多字节突发方式来读取 DS1302 的片内 RAM, 并将数据保存在数组中, 接着将各个数据输出显示。其程序代码示例如下:

```

void BurstReadRAM()
{
    uchar DS1302RAM[31];           //RAM 数组
    uchar i;

    ResetDS1302();                 //复位 DS1302
    WriteByteDS1302(0xFF);         //多字节方式读取 RAM
    for (i=0; i<31; ++i)
    {
        DS1302RAM[i] = ReadByteDS1302(); //保存数据到 RAM 数组
    }
    ResetDS1302();                 //复位 DS1302

    printf("\nDS1302 RAM Data;"); //输出片内 RAM 的数据

    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
        DS1302RAM[0], DS1302RAM[1], DS1302RAM[2], DS1302RAM[3],
        DS1302RAM[4], DS1302RAM[5], DS1302RAM[6], DS1302RAM[7]);
    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
        DS1302RAM[8], DS1302RAM[9], DS1302RAM[10],
        DS1302RAM[11],
        DS1302RAM[12], DS1302RAM[13], DS1302RAM[14],
        DS1302RAM[15]);
    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
        DS1302RAM[16], DS1302RAM[17], DS1302RAM[18],
        DS1302RAM[19],
        DS1302RAM[20], DS1302RAM[21], DS1302RAM[22],
        DS1302RAM[23]);
    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
        DS1302RAM[24], DS1302RAM[25], DS1302RAM[26],
        DS1302RAM[27],
        DS1302RAM[28], DS1302RAM[29], DS1302RAM[30]);
}

```

```
        printf("\nEND RAM Data;");
    }
```

30.2.12 多字节突发方式写入 RAM 函数

多字节突发方式写入 RAM 函数 BurstWriteRAM 用于对 DS1302 的片内 RAM 执行写操作。该函数中，首先获取 RAM 数据，然后采用多字节突发方式来写入 RAM 数据。其程序代码示例如下：

```
void BurstWriteRAM()
{
    uchar Data;
    uchar i;

    printf("\nWrite Ram DATA (HEX):");
    scanf("%bx", &Data); //获取 RAM 数据

    ResetDS1302(); //复位 DS1302
    WriteByteDS1302(0xfe); //多字节方式写入 RAM
    for (i=0; i<31; ++i) //循环写入
    {
        WriteByteDS1302(Data);
    }
    ResetDS1302(); //复位 DS1302
}
```

当程序编写完毕后，便可以编译仿真。如果编译通过，可以将程序写入 AT89S52 单片机中执行，通过串口来控制实时时钟芯片 DS1302 工作。

30.3 小结

本章详细介绍了实时时钟芯片 DS1302 的命令字节和数据格式，并介绍了两种数据传输方式，即单字节传输方式和多字节突发传输方式。本章通过一个具体的实例，讲解了如何使用 8051 单片机来实现对 DS1302 的控制，在该实例中分别采用了单字节传输方式和多字节传输方式对 DS1302 的时钟寄存器及 RAM 进行操作。实时时钟常用于需要时间设定和显示的场合，在单片机应用系统中很常见，读者熟练掌握本章内容，可以轻松实现单片机的时钟显示扩展。

第 31 章 静态 RAM 存储器应用

由于单片机内部存储器容量的限制，在单片机应用系统中，经常需要外扩程序存储器或数据存储器。外部扩展的程序存储器可以用于复杂系统的程序代码的存储，而外部扩展的数据存储器可以用于保存采集或传输的数据。

最常用的存储器为可读写的 RAM 存储器，包括静态 RAM 存储器和动态 RAM 存储器两种。其中，静态 RAM 存储器以其接口简单方便、读写速度快等优点，在单片机系统中得到广泛的应用。

本章将详细介绍 RAM 存储器的特点及常用的静态 RAM 存储器 HM628128，还将对串行通信接口进行详细的介绍。本章将通过一个具体的实例，讲解如何通过计算机的串行通信接口及 8051 单片机来实现静态 RAM 的读写。

31.1 静态 RAM 存储器概述

存储器是计算机或单片机的记忆部件。微处理器的 CPU 将执行的程序、处理的数据及中间结果等都存放在存储器中。

目前常用的存储器芯片几乎全部采用半导体存储器。其有以下两个指标。

- 存储容量，反映了存储记忆信息的多少；
- 存取时间，反映了工作速度的快慢。

半导体存储器芯片根据应用可分为读写存储器（RAM）和只读存储器（ROM）两大类。其中 RAM 同时支持读和写的操作，因此应用十分广泛。下面主要介绍 RAM 存储器。

31.1.1 RAM 存储器概述

RAM 存储器又称随机存取存储器（Random Access Memory），简称为 RAM，它能够在存储器中任意指定的地方随时写入或读出信息；当电源掉电时，RAM 里的内容即消失。

根据存储单元的工作原理，RAM 又分为静态 RAM 和动态 RAM。

- 静态 RAM 用触发器作为存储单元存放 1 和 0，存取速度快，只要不掉电即可持续保持内容不变。一般静态 RAM 的集成度较低，成本较高。
- 动态 RAM 的基本存储电路为带驱动晶体管的电容。电容上有无电荷状态被分别视为逻辑 1 和 0。随着时间的推移，电容上的电荷会逐渐减少，为保持其内容必须周期性地对其进行刷新（对电容充电）以维持其中所存的数据。

由于动态 RAM 的特点，在硬件系统中需要设置相应的刷新电路来完成动态 RAM 的刷新，这样一来无疑增加了硬件系统的复杂程度，因此在单片机应用系统中一般不使用动态 RAM。

静态 RAM 的基本存储电路为触发器，每个触发器存放一位二进制信息，若干个触发器便组成一个存储单元，再由若干存储单元组成存储器阵列，加上地址译码器和读/写控制电路就组成静态 RAM。与动态 RAM 相比，静态 RAM 无须考虑保持数据而设置的刷新电路，故扩展电

路较简单。但由于静态 RAM 是通过有源电路来保持存储器中的数据，因此，要消耗较多功率，价格也较高。

RAM 存储器内容的存取是以字节为单位的，为了区别各个不同的字节，将每个字节的存储单元赋予一个编号，该编号就称为这个存储单元的地址，存储单元是存储的最基本单位，不同的单元有不同的地址。在进行读写操作时，可以按照地址访问不同的单元。

对于大容量存储的复杂系统，如果单片 RAM 容量不够，可以采用若干 RAM 存储器组合使用。此时，在进行读写操作时，通常仅操作其中一片（或几片），这就存在一个片选问题。RAM 存储器芯片上特设了一条片选信号线，在片选信号线上加入有效电平，芯片即被选中，可进行读/写操作，未被选中的芯片不工作。片选信号仅解决芯片是否工作的问题，而芯片执行读还是写则还需有一根读写信号线，所以芯片上必须设读/写控制线。

下面介绍常用的静态 RAM 存储器芯片 HM628128。

31.1.2 静态 RAM 芯片 HM628128

静态 RAM 芯片 HM628128 的 DIP 封装的引脚示意图，如图 31-1 所示。该芯片为 CMOS 型静态 RAM，内部共有 128K×8bit，即 128KB 的存储空间。

该芯片主要的特点如下所述。

- 高速读写，最快可达到 55ns 的读写速度；
 - 低功耗，工作时为 75mW，数据保持状态下为 10μW；
 - 5V 供电，适合于单片机系统；
 - 引脚电平为 TTL 信号，接口方便。
- HM628128 芯片各个引脚的功能介绍，如下所述。
- A0~A16：地址引脚；
 - I/O0~I/O7：输入/输出引脚；
 - $\overline{\text{CS1}}$ ：片选 1；
 - CS2：片选 2；
 - $\overline{\text{WE}}$ ：写使能；
 - $\overline{\text{OE}}$ ：输出使能；
 - NC：不使用；
 - Vcc：正电源；
 - Vss：芯片地。

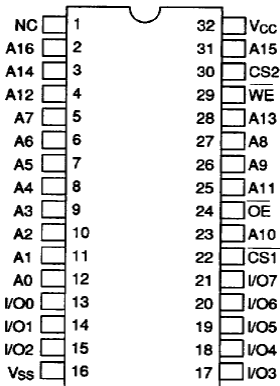


图 31-1 HM628128 引脚示意图

31.1.3 静态 RAM 芯片 HM628128 的读写

静态 RAM 芯片 HM628128 的读写操作由引脚 $\overline{\text{CS1}}$ 、CS2、 $\overline{\text{WE}}$ 和 $\overline{\text{OE}}$ 等来实现。HM628128 读写操作的逻辑功能表，如表 31-1 所示。

表 31-1 逻辑功能表

$\overline{\text{CS1}}$	CS2	$\overline{\text{OE}}$	$\overline{\text{WE}}$	模 式	I/O 引脚	用 途
H	X	X	X	保持	高阻	-
X	L	X	X	保持	高阻	-
L	H	H	H	输出禁止	高阻	-
L	H	L	H	读	输出	读周期
L	H	H	L	写	输入	写周期 1
L	H	L	L	写	输入	写周期 2

其中，H 表示高电平，L 表示低电平，X 表示任意状态。

1. RAM 的读操作

HM628128 的读操作的时序图，如图 31-2 所示。首先需要置 $\overline{CS1}=0$ ， $CS2=1$ ， $\overline{WE}=1$ ，然后 $\overline{OE}=0$ ，则对应地址单元中的数据便输出到数据总线上。

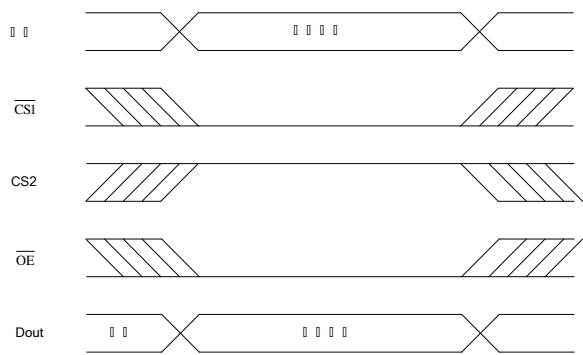


图 31-2 读 HM628128 时序图

2. RAM 的写操作

HM628128 的写操作的时序图，如图 31-3 所示。首先需要置 $\overline{CS1}=0$ ， $CS2=1$ ， $\overline{OE}=1$ ，当 \overline{WE} 从 0 到 1 跳变时，将数据总线上的数据送入对应的地址单元中。

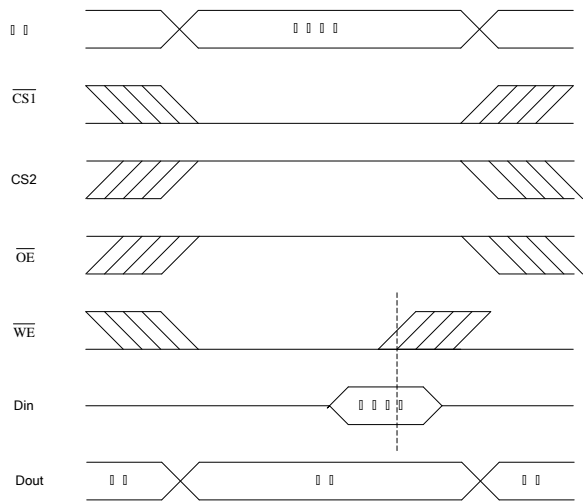


图 31-3 写 HM628128 时序图

31.2 静态 RAM 存储器读写实例

在单片机测控系统中，静态 RAM 存储器常用于程序存储器和数据存储器。下面通过一个具体的实例，讲解如何使用静态 RAM 存储器 HM628128 作为外部数据存储器，以及静态 RAM 存储器的读写。

目前，在测控领域，计算机的应用越来越广泛。使用计算机进行远程控制、数据采集、数据处理等十分方便。计算机在处理这些任务的时候，需要和外部设备进行通信。

由于一般的计算机上都集成有串行接口，可以采用串行接口和外部设备进行通信。51 系列单片机内部也集成有一个全双工的串行异步通信接口，因此，在本实例中采用单片机作为桥梁，实现计算机和外部设备的通信，并完成静态 RAM 存储器的读写。

31.2.1 系统原理

下面通过计算机读写静态 RAM 芯片，这里单片机作为计算机与外部 RAM 存储器通信的桥梁。这个例子涉及知识面比较广。其中用到单片机串行接口的设计、静态 RAM 芯片的读写、计算机串行接口设计等。

首先介绍一下整个系统设计的框架，如图 31-4 所示。其中上位机，即计算机，需要将一批数据保存到外部的静态 RAM 芯片 HM628128 中。上位机和单片机 AT89S52 通过串行通信接口相连，单片机在读写外部静态 RAM 芯片 HM628128 时，采用并行接口方式读写。

整个系统的数据流向是上位机将数据通过串行通信接口逐个发送给单片机，单片机接收到数据后，将数据依次写入 HM628128 的存储单元，直至写满。当数据发送完毕后，单片机再将数据从 HM628128 中读取出来，并送回上位机，以验证 RAM 中数据的正确性。

下面就开始介绍整个系统中的几个关键部分，最后给出完整的电路原理图和上位机程序及单片机程序，以供读者参考。

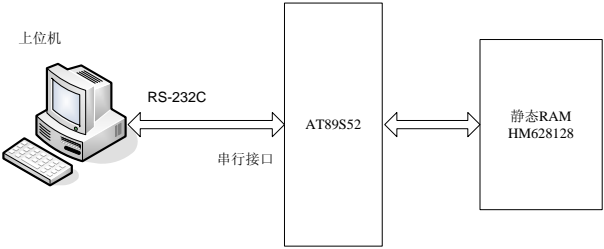


图 31-4 系统结构原理图

31.2.2 串行通信接口概述

目前计算机及绝大多数的测控设备中都配备有异步串行通信接口，主要包括以下三种。

- 基本串行通信接口 RS-232A、RS-232B 和 RS-232C；
- 增强型串行通信接口 RS-449、RS-422、RS-423 和 RS-485；
- 20mA 电流环。

因此，通过这些标准接口，能够方便地将单片机和外部测控设备、计算机或测量仪器等连接起来，构成一个完整的控制、通信系统。下面分别介绍这几种接口标准，其中 RS-232C 是最常用的，本章实例将使用该接口标准和单片机进行通信设计。

1. RS-232C 标准

RS-232C 标准最初是为远程通信连接数据终端 DTE（Data Terminal Equipment）与数据通信设备 DCE（Data Communication Equipment）而制定的，由美国电子工业协会 EIA（Electronic Industries Association）与 BELL 等公司于 1962 年公布，1969 年最后修订。它适合于 0~20000bit/s 范围内的通信。字母 RS 表示 Recommended Standard（推荐标准），232 是识别代号，C 是最后一次修订的标准版本号。

目前，计算机与终端或外设之间的近距离连接，很多都采用 RS-232C 接口，例如串口的鼠标、键盘，以及数据采集系统等。在 RS-232C 标准中，对串行通信接口的信号功能、电气特性和机械特性等都作了明确的规定。由于 RS-232C 在微机系统中的广泛使用，加上 51 系列单片机内部集成的异步串行通信接口，使得它们之间的连接很方便。

在 RS-232C 标准中只规定了采用一对物理连接器,对连接器本身的物理特性没有具体的定义。因此市场上出现了 DB-25、DB-15 和 DB-9 等各种类型的连接器,其引脚的定义也各不相同。目前使用最多的是 DB-9 型连接器,一般的计算机和测控仪器中都使用。DB-9 型 RS-232 引脚结构,如图 31-5 所示。

RS-232C 电缆的长度与传输电容有关,在 RS-232C 标准中,允许的连接电缆约为 15m。如果能使传输电缆的总电容小于 2500pF,则电缆长度可以超过这个限定值。另外,RS-232C 标准所允许的信号传输速率在 0~20000bit/s 范围内,但在实际应用中,常被限制在 19200bit/s 以内,过高的传输速度容易引起传输错误。

EIA RS-232C 标准规定了,在数据终端设备 DTE 和数据通信设备 DCE 之间的串行通信接口信号,对于广泛使用的 DB-9 型连接器,其各个引脚的接口信号,如表 31-2 所示。

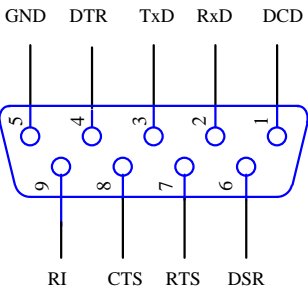


图 31-5 DB-9 型 RS-232 引脚结构

表 31-2 DB-9 型 RS-232C 连接器接口信号

引 脚	名 称	说 明
1	DCD, 载波信号检测 (Data Carrier Detection)	通信设备接收到远程载波
2	RxD, 接收数据 (Received Data)	终端接收串行数据
3	TxD, 发送数据 (Transmitted Data)	终端发送串行数据
4	DTR, 数据终端就绪 (Data Terminal Ready)	终端准备就绪, 可以接收
5	GND, 信号地 (Signal GND)	信号地
6	DSR, 数据设备就绪 (Data Set Ready)	通信设备就绪, 可以接收
7	RTS, 请求发送 (Request To Send)	终端请求通信设备切换到发送状态
8	CTS, 清除发送 (Clear To Send)	通信设备已切换到准备接收
9	RI, 振铃指示 (Ring Indicator)	通信设备通知终端, 通信线路接通

其中,所谓的发送和接收是从数据终端设备 DTE 的角度来定义的。

另外,有的时候会遇到 25 针的 DB-25 型的 RS-232C 连接器,表 31-3 列出了其中各个引脚接口信号的定义。

表 31-3 DB-25 型 RS-232C 连接器接口信号

引 脚	名 称	引 脚	名 称
1	空引脚, 不使用	14	空引脚, 不使用
2	TxD, 发送数据	15	空引脚, 不使用
3	RxD, 接收数据	16	空引脚, 不使用
4	RTS, 请求发送	17	空引脚, 不使用
5	CTS, 清除发送	18	数据接收 (+)
6	DSR, 数据准备好	19	空引脚, 不使用
7	SG, 信号地线路	20	DTR, 数据终端准备好
8	DCD, 载波检测	21	空引脚, 不使用
9	发送返回 (+)	22	RI, 振铃指示
10	空引脚, 不使用	23	空引脚, 不使用
11	数据发送 (—)	24	空引脚, 不使用

12	空引脚，不使用	25	接收返回（—）
13	空引脚，不使用		

在 RS-232C 标准中，对信号的逻辑电平、最高数据传输和各种信号功能作了规定。为了在工业领域中，提供抗干扰能力和增加传送距离，RS-232C 标准中规定的接口信号状态和电压标准不是 TTL 信号。在 RS-232C 中，高电平“1”信号电压的范围为-15V~-3V，低电平“0”信号电压的范围为+3V~+15V。这种高低电平用相反的电压表示，至少有 6V 的电压差，这极大地提高了数据传输的可靠性。

RS-232C 接口标准虽然有广泛的应用，但在现代工业控制及通信领域，其有明显的缺点。主要表现在如下几个方面。

- RS-232C 接口标准只支持单端收发；
- RS-232C 接口标准传输速率低，小于 20000bit/s；
- RS-232C 接口标准抗共模干扰能力差；
- RS-232C 接口标准传输距离短，通常在 15m 以内，在通信干扰小的时候，电缆长度也不超过 60m，很难满足工业控制的要求；
- RS-232C 接口标准没有明确规定连接器，造成了很多互不兼容的连接器，比如 DB-25、DB-15 和 DB-9 等。

为了实现更远距离和更高速度的直接连接，EIA 在 RS-232C 的基础上，制定了更高性能的接口标准，如 RS-449、RS-422A、RS423A 及 RS-485 等。

2. RS-449 标准

RS-449 标准于 1977 年公布，其和 RS-232C 的主要差别是信号在导线上的传输方法不同。在 RS-232C 标准中，传输数据是利用的传输信号和公共地之间的电压差，而在 RS-449 接口标准中，利用的是信号导线之间的信号电压差。

RS-449 接口标准可以实现高达 90000bit/s 的通信速率，如果使用 24-AWG 双绞线，可以实现 1219m 的传输距离。

另外，RS-449 接口不使用调制解调器，也可以实现比 RS-232C 传输速率更高，通信距离更远的串行数据通信。并且，由于使用平衡信号差来传输高速信号，所以其噪声低，可以实现多点或使用公共线通信，故 RS-449 通信电缆可以与多个设备并联。

RS-449 标准中规定了两种标准连接器，一种是 37 引脚，功能相对完善；另一种是 9 个引脚，为简化版本。RS-449 连接器引脚的定义，分别如表 31-4 和表 31-5 所示。

表 31-4 37 引脚的 RS-449 连接器引脚说明

引 脚	信号名称	引 脚	信号名称
1	屏蔽	20	接收公共端
2	信号速率指示器	21	空引脚
3	空引脚	22	发送数据
4	发送数据	23	发送时钟
5	发送同步	24	接收数据
6	接收数据	25	请求发送
7	请求发送	26	接收同步
8	接收同步	27	允许发送
9	允许发送	28	终端正在服务
10	本地回测	29	数据模式

11	数据模式	30	终端就绪
12	终端就绪	31	接收就绪

续表

引 脚	信号名称	引 脚	信号名称
13	接收设备就绪	32	备用选择
14	远距离回测	33	信号质量
15	来话呼叫	34	新信号
16	信号速率选择/频率选择	35	终端定时
17	终端同步	36	备用指示器
18	测试模式	37	发送公共端
19	信号地		

表 31-5 9 引脚的 RS-449 连接器管脚说明

引 脚	信号名称	引 脚	信号名称
1	屏蔽	6	接收器公共端（用于次通道）
2	次通道接收就绪	7	次通道发送请求
3	次通道发送数据	8	次通道发送就绪
4	次通道接收数据	9	发送公共端（用于次通道）
5	信号地		

3. RS-423A 标准

RS-423A 标准于 1978 年 9 月推出，其以非平衡的方式进行数据传输。RS-423A 标准实质上是用差分接收器代替原来的 RS-232C 的单端接收器，如图 31-6 所示。

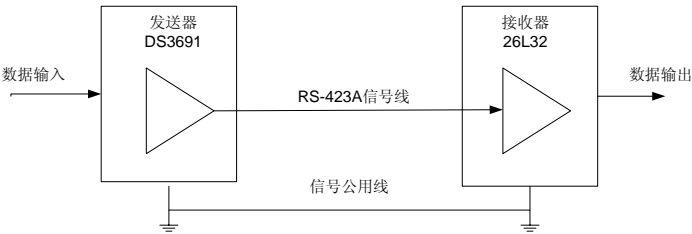


图 31-6 RS-423A 接口标准连接示意图

RS-423A 标准和 RS-232C 相类似，其规定信号参考电平为地。对于 RS-423A 标准与 RS-232C，两者的主要区别在于以下两点。

- RS-423A 标准规定的逻辑高电平“1”的状态必须超过 4V，但不能高于 6V；逻辑低电平“0”的状态必须低于-4V，但不能低于-6V。
- RS-423A 标准中只允许有一个单端发送器，但可以有多多个接收器，接收器采用平衡接收器。

由于 RS-423A 标准采用差动接收，提高了抗共模干扰能力，因此可以达到比 RS-232C 更远的传输距离和更高的传输速率。例如，在最大数据传输速率为 100Kbit/s 时，传输距离为 90m；当最大数据传输速率为 1Kbit/s，传输距离为 1200m。

RS-423A 接口标准也需要进行电平转换，常用的 RS-423A 标准驱动器和接收器为 DS3691 和 26L32。

4. RS-422A 标准

在 RS-423A 标准推出后，接着推出了 RS-422A 标准，这是一种以平衡方式传输的标准。RS-422A 接口标准的连接示意图，如图 31-7 所示。

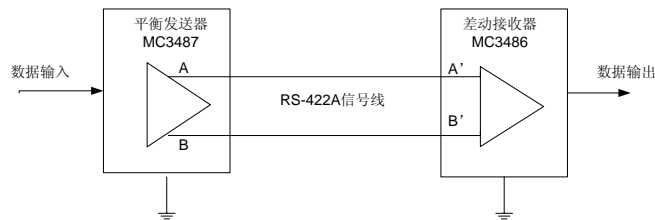


图 31-7 RS-422A 接口标准连接示意图

RS-422A 接口标准采用双端发送和双端接收，传输过程中要用两条线，发送端和接收端分别采用平衡发送器和差动接收器。

RS-422A 接口标准与 RS-232C 标准不同的是，RS-422A 的电气特性对逻辑电平的定义是根据两条传输线之间的电压差决定的。当 AA' 线上的电平比 BB' 线上的电平低 2V 的时候，表示逻辑高电平“1”；当 AA' 线上的电平比 BB' 线上的电平高 2V 的时候，表示逻辑低电平“0”。

由于 RS-422A 接口标准采用了双线传输，以电压差表示信号，两根传输线在传输中受到的干扰基本相同，因此电压差基本不变，这便大大增强了抗共模干扰的能力。

RS-422A 接口标准的最大数据传输速率为 10Mbit/s 时，传输距离为 15m；如果传输速率为 90Kbit/s，则最大的传输距离达到 1200m。

RS-422A 接口电路由发送器、平衡连接电缆、电缆终端负载和接收器组成。RS-422A 接口标准通过平衡发送器将逻辑电平转换成电压差，完成信息发送；然后通过差动接收器，将电压差转换成逻辑电平，实现终端的信息接收。该标准中规定只有一个发送器，但可以有多接收器。RS-422A 的接口标准允许驱动器输出为 $\pm 2V \sim \pm 6V$ ，接收器输入电平可以低至 $\pm 200mV$ 。

RS-422A 标准的接口电路同样需要专用的平衡驱动器/接收器，常用的有 MC3487/MC3486 和 SN75174/SN75175 等。

5. RS-485 标准

RS-485 标准也是一种平衡传输方式的串行接口标准，其和 RS-422A 标准相比，RS-485 标准扩展了 RS-422A 的性能，是一种多发送器的电路标准，允许在双导线上有多个发送器，也允许一个发送器驱动多个负载设备。

RS-485 标准的电路结构，如图 31-8 所示，由一对平衡传输的两端都配置终端电阻，其发送器、接收器及组合收发器都可以挂接在平衡传输线的任意位置，从而实现了数据传输中多个驱动器和接收器公用一条传输线的多点应用。

虽然 RS-485 标准允许电路中出现多个发送器，但 RS-485 仅能工作于半双工方式，即任一时刻只允许一个发送器发送数据，而其他组件只能处于接收状态。

RS-485 标准的特点是抗干扰能力强，传输距离远，速率高。如果采用双绞线传输信号，如果最大传输速率为 10Mbit/s，传输距离为 15m；在最大 100Kbit/s 的传输速率下，可以传输 1200m；如果最大传输速率为 9600bit/s，则传输距离可达 1500m。

RS-485 标准最多允许在平衡电缆上连接 32 个发送器/接收器，特别适用于工业控制领域进行分布管理、联网检测控制等，目前得到很广泛的应用。

6. 20mA 电流环路串行接口

20mA 电流环路串行接口是一种异步串行通信接口标准，在每次发送数据的时候必须以

无电流的起始状态作为每一个字符的起始位，接收端检测到起始位后便开始接收字符数据。在串行通信中，20mA 电流环也是目前一种应用的比较广泛的接口电路，其原理图如图 31-9 所示。

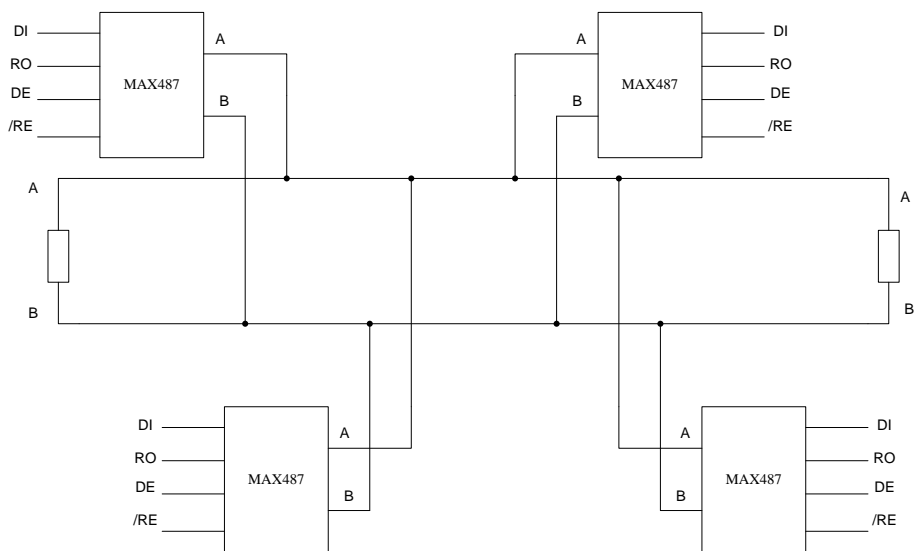


图 31-8 RS-485 接口标准网络的典型应用

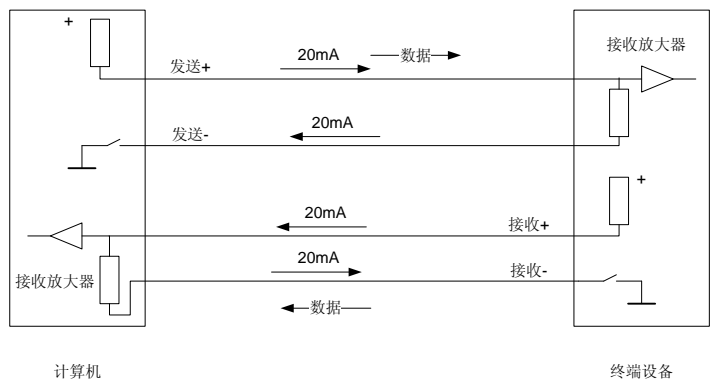


图 31-9 20mA 电流环路原理图

图 31-9 中两根传输线发送+和发送-组成一个输入电流回路，另两根传输线接收+和接收-组成一个输出电流回路。当发送数据时，根据数据的逻辑 0 和 1，使回路有规律地形成通、断状态。

20mA 电流环串行通信接口主要的优点是低阻传输线对电气噪声不敏感，而且比较容易实现光电隔离，适用于长距离的数据通信。

31.2.3 单片机与 RS-232C 的接口

本例中需要采用单片机作为数据通信的桥梁。前面介绍过计算机的 RS-232C 接口的电平范围为-15V~+15V，由于单片机的接口电路为 TTL 信号，因此单片机与具有 RS-232C 标准的串行接口（例如计算机）进行通信的时候，首先要解决的便是电平转换的问题。一般来说，可以选择一些专用的集成电路芯片。

51 单片机开发与应用技术详解

对于单向的串行数据传输，可以选择 MC1489 或 SN75189 芯片来实现 EIA 电平到 TTL 电平的转换，也可以选择 MC1488 或 SN75188 芯片来实现 TTL 电平到 EIA 电平的转换。这些芯片只能实行单方向的电平转换，并且使用时需要提供±12V 的电源电压，这对于 5V 供电的单片机系统来说，不是很方便。

对于双向的串行数据传输，可以选择 MAX232 或 MAX232A 等芯片来实现 EIA 电平和 TTL 电平之间的互换。MAX232 和 MAX232A 只需+5V 供电，该芯片内部集成了电压倍增电路，特别适合于单片机系统，而且该芯片内部含有两对串行收发线路，也便于实现多机通信。MAX232 和 MAX232A 的引脚结构图，如图 31-10 所示。MAX232 和 MAX232A 典型应用，如图 31-11 所示。

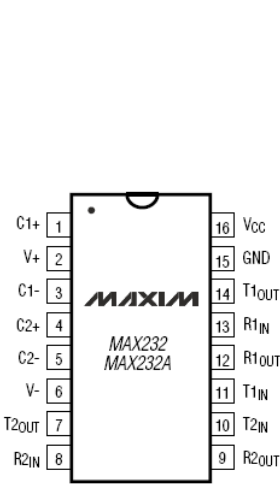


图 31-10 MAX232 和 MAX232A 引脚结构

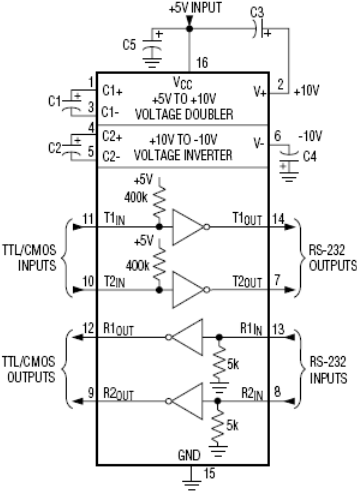


图 31-11 MAX232 和 MAX232A 典型应用

其中，对于 MAX232，各个电容的数值均为 1μF，对于 MAX232A，各个电容的数值均为 0.1μF。

如果采用 MAXIM 公司的 MAX232 作为电平转换器，则单片机与 RS-232C 接口示意图，如图 31-12 所示。

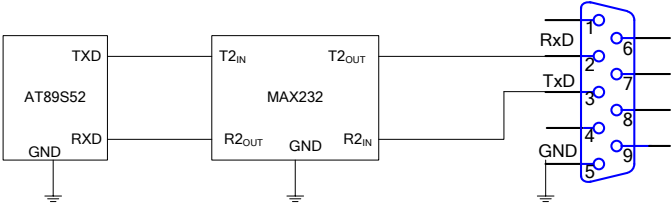


图 31-12 单片机与 RS-232C 接口示意图

31.2.4 系统电路原理图

系统电路原理图，如图 31-13 所示。

电路中所用的元件清单，如表 31-6 所示。

表 31-6 元件表

元 器 件	数 值	数 量
U1	AT89S52	1 个

31.2.5 建立项目

这里采用 Keil C51 语言进行程序设计。首先在 Keil μ Vision3 集成开发环境中建立项目，具体操作步骤如下：

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 31-14 所示。
- (3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 31-15 所示。单击“是”按钮，完成工程的建立。

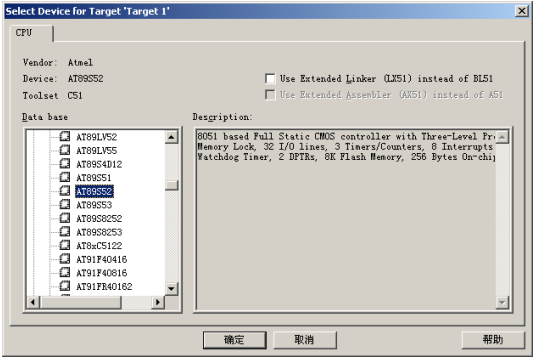


图 31-14 选择单片机 AT89S52



图 31-15 “ μ Vision3”对话框

- (4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

31.2.6 主程序设计

AT89S52 主程序的流程图，如图 31-16 所示。

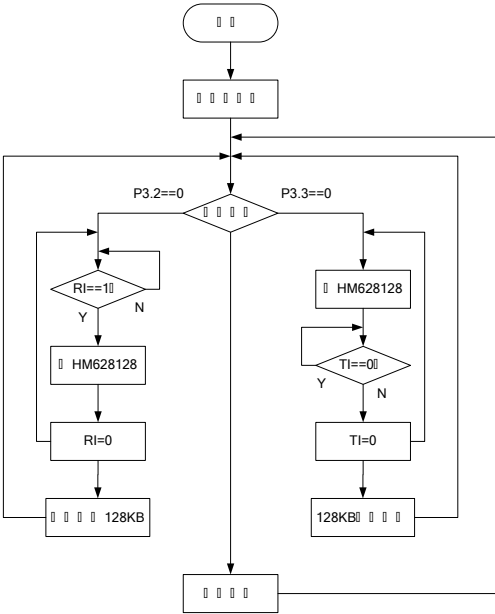


图 31-16 主程序流程图

单片机 AT89S52 负责完成和计算机的串口通信及对外部静态 RAM 存储器 HM628128 的读写操作。其中，程序开始时首先初始化串口，设置和计算机通信的波特率为 4800bit/s。在 while 主循环中，通过判断 P3.2 引脚和 P3.3 引脚的电平，来决定是接收计算机的串行数据还是向计算机发送串行数据。

如果 P3.2 为低电平，则计算机通过 RS-232C 接口向单片机发送数据，AT89S52 接收串行数据，然后通过并行端口将数据逐个写入静态 RAM 存储器 HM628128。

如果 P3.3 为低电平，则单片机逐个读取静态 RAM 存储器 HM628128 中的数据，然后通过串行接口发送给计算机。

在对静态 RAM 存储器 HM628128 进行读写操作时，可以参照前面介绍的读写操作时序。

AT89S52 主程序的示例代码如下：

```
#include <reg52.h>
#include <stdio.h>

typedef unsigned char  BYTE;           //宏定义
typedef unsigned short WORD;
typedef unsigned long  DWORD;

#define MSB(word)      (BYTE)((WORD)(word) >> 8) & 0xff)
#define LSB(word)      (BYTE)((WORD)(word) & 0xff)

sbit P3_2=P3^2;                        //计算机发送，单片机接收的标志
sbit P3_3=P3^3;                        //单片机发送，计算机接收的标志
sbit P3_4=P3^4;                        //HM628128 最高位地址
sbit P3_6=P3^6;                        //HM628128 写使能
sbit P3_7=P3^7;                        //HM628128 读使能

bit AT89S52Send=0;                    //单片机发送，计算机接收的软件标志位

void main(void)                        //主程序
{
    unsigned int count;
    DWORD t;
    int i,j;
    count=0;                           //初始化变量
    t=0;

    SCON=0x50;                          //串口模式 1，允许接收
    TMOD|=0x20;                          //初始化 T1 为定时功能，模式 2
    PCON|=0x80;                          //设置 SMOD=1
    TL1=0xF4;                            //波特率 4800bit/s，初值
    TH1=0xF4;
    IE|=0x90;                            //中断
    TR1=1;                               //启动定时器
    P3_6=1;                              //写使能
    P3_7=1;                              //读使能
    AT89S52Send=1;                       //软件标志

    while(1)                             //主循环
    {
        while(P3_2==0)                   //计算机发送，单片机接收
```

51 单片机开发与应用技术详解

```
{
    if (RI) //如果单片机接收到数据
    {
        P0=LSB(count); //取低 8 位
        P2=MSB(count); //取高 8 位
        if (t<=0xFFFF)
        {
            P3_4=0; //最高位
        }
        else
        {
            P3_4=1;
        }
        count++; //计数递增
        t++;

        P3_6=0; //HM628128 写使能
        RI=0; //RI 清零
        P1=SBUF; //将计算机发送的数据写入 P1 口
        P3_6=1; //HM628128 禁止写
    }

    AT89S52Send=1; //软件标志位
}

while (P3_3==0 & AT89S52Send==1) //单片机发送，计算机接收
{
    for (j=0;j<=0xFF;j++) //读 0000H~FFFFH
    {
        P3_4=0;
        for (i=0;i<=0xFF;i++)
        {
            P0=i;
            P2=j;
            P3_7=0; //HM628128 读使能
            SBUF=P1; //将读到的数据写入 SBUF，启动串行发送
            while (TI==0); //等待发送完毕
            TI=0; //TI 清零
            P3_7=1; //HM628128 禁止读
        }
        P3_7=1;
    }

    for (j=0;j<=0xFF;j++) //读 10000H~1FFFFH
    {
        P3_4=1;
        for (i=0;i<=0xFF;i++)
        {
            P0=i;
            P2=j;
            P3_7=0; //HM628128 读使能
            SBUF=P1; //将读到的数据写入 SBUF，启动串行发送
            while (TI==0); //等待发送完毕
            TI=0; //TI 清零
```

```

        P3_7=1;                //HM628128 禁止读
    }
    P3_7=1;
}
AT89S52Send=0;                //读标志清零
}
}
}

```

程序输入完毕，便可以进行编译仿真，并下载到 AT89S52 单片机中执行。

31.2.7 系统运行

上位机的程序可以采用 VC 或 VB 进行开发，由于 VC 方面的程序设计不是本书的重点，这里不再具体介绍程序的设计步骤，用户可以参阅其他介绍 VC 或 VB 等介绍串口程序设计的书籍。

上位机软件的主要功能是选择特定的文件，并将其通过串行口发送给单片机，同时还可以接收单片机返回的数据。

整个系统的运行可以实现 RAM 数据的读写，其中系统写 RAM 存储器的操作步骤如下：

(1) 首先，硬件上置 P6 开关不闭合，即复位状态，不进行任何读写操作。

(2) 连接硬件电路至计算机的某个串行口，并使硬件上电。

(3) 在上位机软件中，并选择刚才连接的串行口。由于系统规定了串行口的波特率固定为 4800bit/s，因此上位机软件中同样需要设置波特率为 4800bit/s。

(4) 将开关 P6 置 M1，此时，单片机进入接收计算机串行数据并写 HM628128 状态。

(5) 在上位机程序中，首先选择一个 128KB 的测试文件，假设命名为“TestDataFile.txt”。

(6) 在上位机程序中，执行发送功能，开始向单片机发送数据。

当 128KB 数据全部发送完毕时，将开关 P6 置为不闭合即可完成 RAM 存储器的写操作。

整个系统读 RAM 存储器的操作步骤如下：

(1) 首先，硬件上置 P6 开关不闭合，即复位状态，不进行任何读写操作。

(2) 连接硬件电路至计算机的某个串行口，并使硬件上电。

(3) 在上位机软件中，并选择刚才连接的串行口。由于系统规定了串行口的波特率固定为 4800bit/s，因此上位机软件中同样需要设置波特率为 4800bit/s。

(4) 将开关 P6 置 M2，此时单片机自动进入读 HM628128 状态，并将读到的数据返回给计算机。

(5) 将读到的数据保存在一个文件中，例如命名为“SaveSerialPort.txt”。

(6) 当 RAM 数据全部读取完毕时，将开关 P6 置为不闭合即可完成 RAM 存储器的读操作。

如果为了测试程序运行是否正常，可以选择一个 128KB 的测试文件，将其写入静态 RAM 存储器。当数据写入完毕的时候，将 RAM 中的数据再次读出。此时，使用 UltraEdit 的文件比较器，分别打开发送的测试文件和接收到的文件，进行比较，如图 31-17 所示。

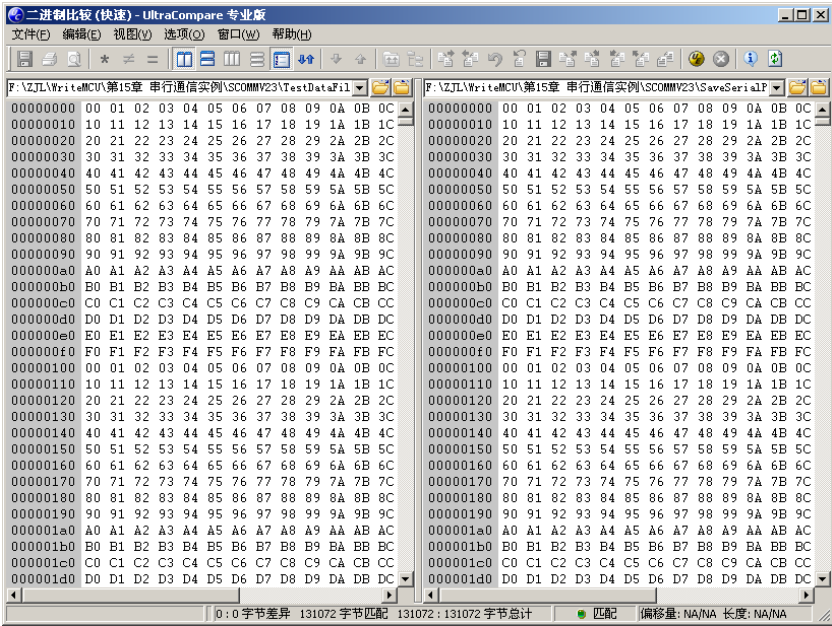


图 31-17 数据比较

如果两个文件完全一致，则表示程序对静态 RAM 存储器 HM628128 的读写完全正确。

31.3 小结

本章详细讲述了 RAM 存储器的种类和特点，并重点讲解了应用最为广泛的静态 RAM 存储器。本章还对一款常用的静态 RAM 存储器 HM628128 进行了详细介绍。最后，通过一个综合的实例介绍了静态 RAM 存储器的读写。实例中使用了计算机的串行通信接口，以及单片机的串行接口设计。通过本章的讲解，读者可以掌握单片机系统中静态 RAM 存储器的读写及计算机串行的应用。

第 32 章 道路交通灯控制系统

实时多任务操作系统（RTOS）常用于处理复杂的多任务控制系统。8051 单片机支持典型的 RTX-51 实时多任务操作系统。通过 RTX-51 的任务管理工具，可以使单个 51 系统微处理器可以管理多个任务或进程。目前，51 系列单片机最常用的是 RTX-51 Tiny 实时多任务操作系统。本章将通过一个道路交通灯控制系统，来讲解 RTX-51 Tiny 实时多任务操作系统的设计。

32.1 交通灯控制系统概述

道路交通灯也就是常说的红绿灯，这是最常见的一种实时控制系统，在一般的道路交叉口都可以看到。道路交通灯的控制综合了一般测控系统常用的功能，包括按键输入、时钟控制、显示及串口通信等。下面介绍道路交通灯的基本原理。

32.1.1 道路交通灯概述

典型的道路交通灯示意图，如图 32-1 所示。其中主干道为双向的交通线路，和其垂直的辅路可供行人行走。主干道上的红绿灯指挥车辆的行驶，辅路上的红绿灯指挥行人的通过与禁止。行人按钮用于行人过马路的申请，当按下该按钮后，主干道变为红灯，禁止车辆通过，此后行人路灯变为绿灯，行人便可以通过马路。

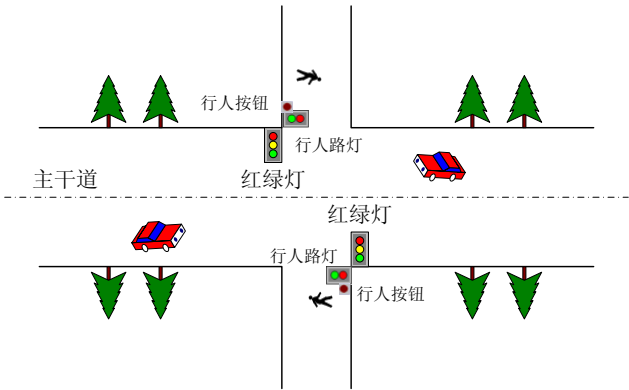


图 32-1 道路交通灯示意图

32.1.2 交通灯控制系统

道路交通灯使用一般的单进程序很难简捷有效地实现，如果采用实时多任务系统则可以很容易达到很好的效果。在一个用户定义的时间段里，交通灯受系统控制。在规定时间段之外，黄灯闪烁。如果一个行人按下了行人按钮，交通灯立即进入行人过马路状态，表示行人需要过马路。否则，交通灯持续不断地工作。

如果采用实时多任务操作系统，则道路交通灯控制系统至少应该包括如下几个任务。

- 任务 0：初始化系统，并且启动所有其他的任务；
- 任务 1：完成交通灯控制器的命令处理；

51 单片机开发与应用技术详解

- 任务 2：控制系统时钟；
- 任务 3：如果时间在活跃的时间段之外，使黄色灯闪烁；
- 任务 4：当时间在活跃的时间段（在开始和结束时间之间）内以后，控制交通灯的交变显示；
- 任务 5：读取行人按钮是否按下，并且执行相应的处理。

在实际设计中，为了显示及系统控制的需要，程序中还应该包括命令输入及命令处理等功能。本章将采用 51 系列单片机，通过 RTX-51 Tiny 实时多任务操作系统来实现道路交通灯控制系统。

32.2 交通灯控制系统原理图

道路交通灯控制系统原理图，如图 32-2 所示。该电路中所使用的元器件清单，如表 32-1 所示。

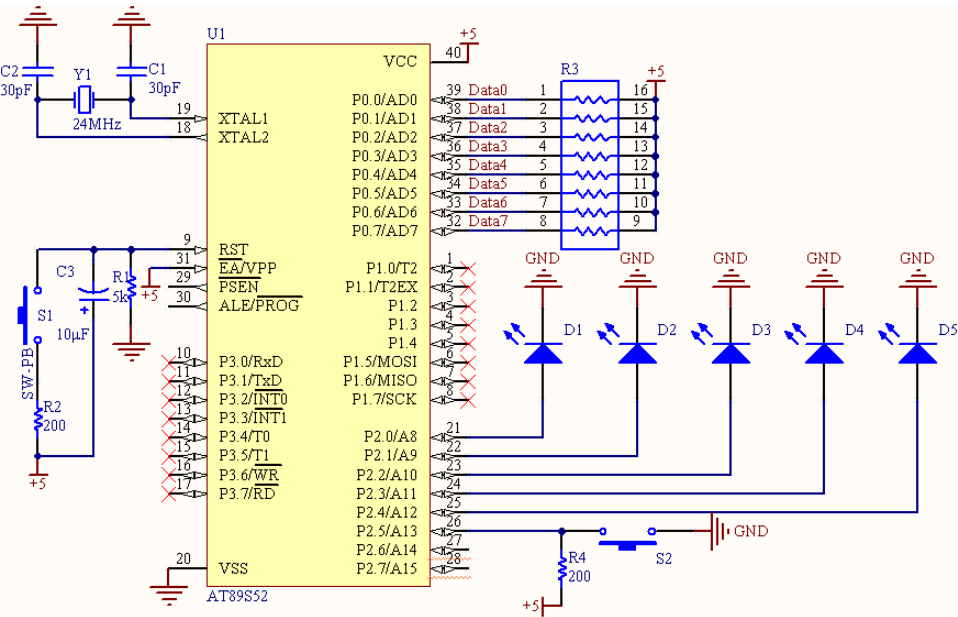


图 32-2 交通灯控制系统原理图

表 32-1 元器件列表

元 器 件	数 值	数 量
U1	AT89S52	1 个
U2	ADG201HS	1 个
C1、C2	30pF	2 个
C3	10μF 电解	1 个
S1、S2	按键开关	2 个
R1	5k Ω	1 个
R2、R4	200 Ω	2 个
R3	10k Ω 排阻	1 个

续表

元 器 件	数 值	数 量
D1~D5	发光二极管	5 个
Y1	24MHz	1 个

该电路中使用 P2 端口作为红绿灯显示及按钮输入控制，其示例如下：

- P2.0 引脚作为绿灯输出；
- P2.1 引脚作为黄灯输出；
- P2.2 引脚作为红灯输出；
- P2.3 引脚作为行人红灯输出；
- P2.4 引脚作为行人绿灯输出；
- P2.5 引脚作为行人按钮输入。

32.3 多任务交通灯控制系统程序

下面在 Keil μ Vision3 集成开发环境中，使用 Keil C51 语言进行道路交通灯控制系统的设计。其中采用了 RTX-51 Tiny 的实时多任务操作系统。

32.3.1 建立项目

首先在 Keil μ Vision3 集成开发环境中建立项目，具体操作步骤如下：

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 32-3 所示。
- (3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 32-4 所示。单击“是”按钮，完成工程的建立。

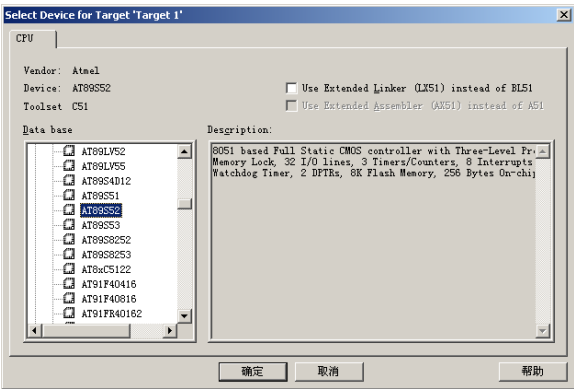


图 32-3 选择单片机 AT89S52

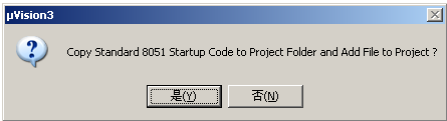


图 32-4 “ μ Vision3”对话框

- (4) 选择“Project”→“Options for Target ‘Target 1’”命令，此时弹出“Options for Target ‘Target 1’”对话框，如图 32-5 所示。
- (5) 在“Target”选项卡中，打开“Operating”下拉菜单，从中可以选择 RTX-51 Tiny 实时多任务操作系统。
- (6) 单击“确定”按钮，关闭对话框，完成目标操作系统的指定。
- (7) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

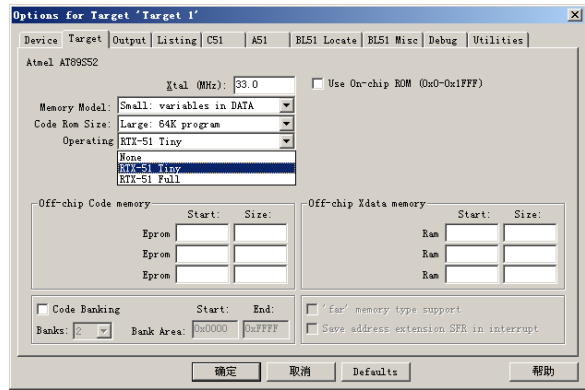


图 32-5 选择目标操作系统

32.3.2 多任务划分及程序设计

使用 RTX-51 Tiny 实时多任务操作系统进行交通灯控制,整个系统将被分成如下几个任务。

- 任务 0: 用于初始化串口, 并且启动所有其他的任务;
- 任务 1: 用于完成交通灯控制器的命令处理, 并负责控制和处理接收到的串行命令;
- 任务 2: 用于控制系统时钟;
- 任务 3: 如果时间在活跃的时间段之外, 黄色灯闪烁;
- 任务 4: 当时间落在活跃的时间段 (在开始和结束时间之间) 内以后, 控制交通灯的交错变化;
- 任务 5: 读取行人按钮是否按下, 如果按钮按下则向任务 4 发送信号;
- 任务 6: 如果在串行指令里遇到了 ESC 字符, 则向任务 1 发送一个信号, 并且终止显示命令。

下面将介绍 RTX-51 实时多任务程序。首先, 需要将头文件 rtx51tny.h 添加到程序中, 并且分别定义各个任务的任务号, 以及控制交通灯的单片机引脚。预定义部分还声明了时间结构用于时间显示及设定。程序预定义部分的代码示例如下:

```
#include <reg52.h> //头文件
#include <rtx51tny.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

code char sysmenu[] = //系统命令菜单
    "\n"
    "+*** TRAFFIC LIGHT CONTROLLER using AT89S52 and RTX-51 Tiny ***+\n"
    "+ 命令 ----+ 格式 -----+ 功能 -----+ \n"
    "| Display | D | 显示时间 | \n"
    "| Time | T hh:mm:ss | 设置时钟时间 | \n"
    "| start_time | S hh:mm:ss | 设置起始时间 | \n"
    "| end_time | E hh:mm:ss | 设置结束时间 | \n"
    "+-----+-----+-----+ \n";

idata char inline[16]; //存储输入命令行

#define uchar unsigned char //宏定义

extern getline (char idata *, char); //外部函数声明, 用于行输入
```



```

extern init_serial ();                                //外部函数声明，用于初始化串行口

#define INIT      0                                  //任务 0： 初始化系统
#define COMMAND   1                                  //任务 1： 命令处理
#define CLK       2                                  //任务 2： 时钟控制
#define BLINKING  3                                  //任务 3： 黄色灯闪烁
#define LIGHTS    4                                  //任务 4： 交通灯工作
#define buttonREAD 5                                //任务 5： 读取行人按钮
#define GET_ESC   6                                  //任务 6： 退出

#define ESC 0x1B                                       //定义 ESC 键代码

struct time {                                          //定义时间结构
    uchar hour;                                       //小时
    uchar min;                                       //分
    uchar sec;                                       //秒
};

struct time clk_time = { 0, 0, 0 };                  //存储时钟的时间值
struct time start_time = { 6, 30, 0 };               //开始时间
struct time end_time = { 20, 30, 0 };                //结束时间

struct time ztime;                                     //输入时间暂存

bit escape;                                           //ESC 键输入标志
bit display_time = 0;                                //信号命令状态显示时间标志

sbit green = P2^0;                                    //绿灯
sbit yellow = P2^1;                                  //黄灯
sbit red = P2^2;                                      //红灯
sbit stop = P2^3;                                    //行人停止灯
sbit walk = P2^4;                                    //行人通过灯
sbit button = P2^5;                                  //行人按钮

```

1. 任务 0

任务 0 用于初始化串口，并且启动所有其他的任务。由于初始化只需要进行一次，任务 0 将自己删除自己。任务 0 的代码示例如下：

```

init () _task_ INIT                                  //任务 0，程序起始处
{
    init_serial ();                                  //初始化单片机串行口
    os_create_task (CLK);                            //启动任务 2
    os_create_task (COMMAND);                        //启动任务 1
    os_create_task (LIGHTS);                         //启动任务 4
    os_create_task (buttonREAD);                    //启动任务 5
    os_delete_task (INIT);                          //停止任务 0
}

```

51 单片机开发与应用技术详解

2. 任务 1

任务 1 用于完成交通灯控制器的命令处理，并负责控制和处理接收到的串行命令。其中，switch 语句对输入的命令进行识别。任务 1 的代码示例如下：

```
void command () _task_ COMMAND //任务 1
{
    uchar i;

    printf (systemenu); //系统命令菜单
    while (1)
    {
        printf ("\nCommand: "); //显示命令提示符
        getline (&inline, sizeof (inline)); //获得命令行输入

        for (i = 0; inline[i] != 0; i++) //将命令字符串转换为大写
        {
            inline[i] = toupper(inline[i]);
        }

        for (i = 0; inline[i] == ' '; i++); //跳过字符串中的空格符

        switch (inline[i]) //命令处理
        {
            case 'D': //显示时间命令
                printf ("start_time=: %02bd:%02bd:%02bd "
                    "end_time=: %02bd:%02bd:%02bd\n",
                    start_time.hour, start_time.min, start_time.sec,
                    end_time.hour, end_time.min, end_time.sec);
                printf ("Input ESC to abort\r");

                os_create_task (GET_ESC); //在显示任务中检查 ESC 键
                display_time = 1; //显示时间标志置 1
                escape = 0; //清除 ESC 标志
                os_clear_signal (COMMAND);

                while (!escape) //如果没有输入 ESC
                {
                    printf ("Time: %02bd:%02bd:%02bd\r",
                        clk_time.hour, clk_time.min, clk_time.sec);
                    os_wait (K_SIG, 0, 0); //等待
                }

                os_delete_task (GET_ESC); //删除任务
                display_time = 0; //显示时间标志置 0
                printf ("\n\n"); //输出换行符
                break;

            case 'T': //设置时间命令
                if (readtime (&inline[i+1])) //获取输入时间
                {
                    clk_time.hour = ztime.hour;
                    clk_time.min = ztime.min;
                    clk_time.sec = ztime.sec;
                }
                break;
        }
    }
}
```

```

case 'S':                                //设置开始时间命令
    if (readtime (&inline[i+1]))        //获取输入时间
    {
        start_time.hour = ztime.hour;
        start_time.min  = ztime.min;
        start_time.sec  = ztime.sec;
    }
    break;

case 'E':                                //设置结束时间命令
    if (readtime (&inline[i+1]))        //获取输入时间
    {
        end_time.hour = ztime.hour;
        end_time.min  = ztime.min;
        end_time.sec  = ztime.sec;
    }
    break;

default:                                //无效指令
    printf("Error!");
    printf (sysmenu);                  //显示系统命令菜单
    break;
}
}

```

在该程序中，分别设置了如下 4 条命令。

- 显示时间命令 D;
- 设置时间命令 T，格式为 hh:mm:ss;
- 设置开始时间命令 S，格式为 hh:mm:ss;
- 设置结束时间命令 E，格式为 hh:mm:ss。

该函数中还调用了读取时间函数 `readtime`，用于获取输入的小时、分钟和秒时间。读取时间函数 `readtime` 的程序代码示例如下：

```

bit readtime (char idata *buffer)        //读取时间函数
{
    uchar args;

    ztime.sec = 0;                        //初始化
    args = sscanf (buffer, "%d:%d:%d",    //扫描输入，并保存至变量
        &ztime.hour,
        &ztime.min,
        &ztime.sec);

    if (ztime.hour > 23 || ztime.min > 59 ||    //检查输入时间值是否有效
        ztime.sec > 59 || args < 2 || args == EOF)
    {
        printf ("\nINVALID TIME FORMAT! \n"); //输出错误信息
        return (0);
    }
    return (1);
}

```

3. 任务 2

任务 2 用于控制系统时钟。在任务 2 中执行无限循环，用于一直计算小时、分钟和秒的数值。当需要显示时间时，则向任务 1 发送信号执行显示命令。任务 2 的代码示例如下：

```

void clock () _task_ CLK                //任务 2

```

```
{
while (1)
{
    if (++clk_time.sec == 60)                //计算秒值
    {
        clk_time.sec = 0;
        if (++clk_time.min == 60)           //计算分钟值
        {
            clk_time.min = 0;
            if (++clk_time.hour == 24)       //计算小时值
            {
                clk_time.hour = 0;
            }
        }
    }
}
if (display_time)                          //显示时间
{
    os_send_signal (COMMAND);               //向任务 1 发送信号
}
os_wait (K_IVL, 100, 0);                   //等待 1 秒
}
}
```

4. 任务 3

任务 3 用于系统时间运行在活跃的时间段之外时，控制黄色灯闪烁。开始时间和结束时间之间为活跃时间。在程序中使用 **while** 循环来使黄色交通灯闪烁。程序中还判断闪烁时间段是否结束，如果结束则启动任务 4，并删除本任务。任务 3 的代码示例如下：

```
void blinking () _task_ BLINKING           //任务 3
{
    red    = 0;                            //熄灭红灯
    yellow = 0;                            //熄灭黄灯
    green  = 0;                            //熄灭绿灯
    stop   = 0;                            //熄灭行人红灯
    walk   = 0;                            //熄灭行人绿灯

    while (1)                              //闪烁循环
    {
        yellow = 1;                        //点亮黄灯
        os_wait (K_TMO, 30, 0);            //等待超时
        yellow = 0;                        //熄灭黄灯
        os_wait (K_TMO, 30, 0);            //等待超时
        if (signalon ())                   //如果闪烁时间段结束
        {
            os_create_task (LIGHTS);        //启动任务 4
            os_delete_task (BLINKING);      //停止任务 3
        }
    }
}
```

该程序中，使用了 **signalon** 函数来检查系统时钟是否处在开始时间和结束时间之间。如果该函数返回 0 则表示在活跃时间之外，如果返回值为 1 则表示在活跃时间之内。**signalon** 函数的代码示例如下：

```

bit signalon ()                                //检查系统时钟是否处在开始时间和结束时间之间
{
    if (memcmp (&start_time, &end_time, sizeof (struct time)) < 0)
    {
        if (memcmp (&start_time, &clk_time, sizeof (struct time)) < 0 &&
            memcmp (&clk_time, &end_time, sizeof (struct time)) < 0)
            return (1);                          //红绿灯处于活跃状态
    }
    else
    {
        if (memcmp (&end_time, &clk_time, sizeof (start_time)) > 0 &&
            memcmp (&clk_time, &start_time, sizeof (start_time)) > 0)
            return (1);
    }
    return (0);                                //信号结束, 开始闪烁黄灯
}

```

5. 任务 4

任务 4 用于系统时间运行在活跃的时间段之内时, 控制红绿灯循环工作。开始时间和结束时间之间的为活跃时间。在程序中使用 **while** 循环来使红绿灯交通灯交错工作。程序中使用 **os_wait** 函数来实现灯亮的时间。任务 4 的代码示例如下:

```

void lights () _task_ LIGHTS                    //任务 4
{
    red    = 1;                                //点亮红灯
    yellow = 0;                                //熄灭黄灯
    green  = 0;                                //熄灭绿灯
    stop   = 1;                                //点亮行人红灯
    walk   = 0;                                //熄灭行人绿灯

    while (1)                                  //主循环
    {
        os_wait (K_TMO, 30, 0);                //等待超时
        if (!signalon ())                      //如果系统时间处在活跃时间之外
        {
            os_create_task (BLINKING);          //启动任务 3
            os_delete_task (LIGHTS);            //停止任务 4
        }
        yellow = 1;
        os_wait (K_TMO, 30, 0);                //等待超时

        red    = 0;
        yellow = 0;
        green  = 1;                            //为主干道点亮绿灯

        os_clear_signal (LIGHTS);
        os_wait (K_TMO, 30, 0);                //等待超时
        os_wait (K_TMO + K_SIG, 310, 0);        //等待超时和信号

        yellow = 1;
        green  = 0;
        os_wait (K_TMO, 30, 0);                //等待超时

        red    = 1;                            //为主干道点亮红灯
        yellow = 0;
    }
}

```

```
os_wait (K_TMO, 30, 0);           //等待超时
stop   = 0;                       //为行人点亮绿灯，放行
walk   = 1;

os_wait (K_TMO, 100, 0);          //等待超时
stop   = 1;                       //为行人点亮红灯
walk   = 0;
}
}
```

6. 任务 5

任务 5 用于读取行人按钮是否按下，如果按钮按下则向任务 4 发送信号。该任务通过 while 循环来读取行人按钮，当有行人按下时候便通过 os_send_signal 函数向任务 4 发送信号，从而放行行人。任务 5 的代码示例如下：

```
void buttonread () _task_ buttonREAD //任务 5
{
    while (1)
    {
        if (button) //若行人按钮按下
        {
            os_send_signal (LIGHTS); //向任务 4 发送信号
        }
        os_wait (K_TMO, 3, 0); //等待超时
    }
}
```

7. 任务 6

任务 6 用于检查是否有 ESC 输入。如果在串行指令里检测到 ESC 字符，则向任务 1 发送一个信号，并且终止显示命令。任务 6 的代码示例如下：

```
void get_escape () _task_ GET_ESC //任务 6
{
    while (1)
    {
        if (_getkey () == ESC) //如果 ESC 键输入，置 escape 标志
            escape = 1;
        if (escape) //如果有 escape 标志，向任务 1 发送信号
        {
            os_send_signal (COMMAND);
        }
    }
}
```

32.3.3 串行通信函数

串行通信函数用于处理串行口的中断数据通信。其函数包含在 SERIAL_Communication.c 文件中。这里包含基本的串行口初始化、字符输入输出以及串行中断处理等。下面在介绍各个函数之前，首先给出程序需要用到的一些头文件及预定义，示例如下：

```
#include <reg52.h> //头文件
#include <rtx51tny.h>

#define uchar unsigned char //宏定义

#define OUTBUF_LEN 8 //串行口发送缓冲区的长度
```

```

uchar outbuf_start;           // 串行口发送缓冲区的起点
uchar outbuf_end;             // 串行口发送缓冲区的终点
idata char outbuf[OUTBUF_LEN]; // 串行口发送缓冲区
uchar out_task = 0xff;        // 串行口输出任务的任务号

#define INBUF_LEN 8           // 串行口接收缓冲区的长度
uchar inbuf_start;           // 串行口接收缓冲区的起点
uchar inbuf_end;             // 串行口接收缓冲区的终点
idata char inbuf[INBUF_LEN]; // 串行口接收缓冲区
uchar in_task = 0xff;         // 串行口输入任务的任务号

#define CTRL_Q 0x11
#define CTRL_S 0x13

bit sendfull;                // 发送缓冲区满标志位
bit sendactive;              // 发送激活标志位
bit sendstop;                // 发送停止标志位

```

下面分别介绍这里用到的函数。

1. putbuf 函数

putbuf 函数用于向 SBUF 或发送缓冲区写入字符。程序中首先判断发送缓冲区满标志位 sendfull，如果发送缓冲区未满足则直接启动发送字符，否则向发送缓冲区传送字符。putbuf 函数的代码示例如下：

```

void putbuf (char c)
{
    if (!sendfull)                // 判断发送缓冲区满标志位
    {
        if (!sendactive && !sendstop) // 缓冲区未满足，发送器未处于活动态
        {
            sendactive = 1;
            SBUF = c;                // 直接发送第一个字符到 SBUF，并启动发送
        }
        else
        {
            outbuf[outbuf_end++ & (OUTBUF_LEN-1)] = c; // 向定义的发送缓冲区传送字符
            if (((outbuf_end ^ outbuf_start) & (OUTBUF_LEN-1)) == 0)
                sendfull = 1;        // 缓冲区满标志位置位
        }
    }
}

```

2. putchar 函数

putchar 函数用于通过串行口发送字符，该函数的返回值为发送的字符。该函数中使用 while 循环等待 sendfull/缓冲区满标志，然后调用 putbuf 函数发送字符，并通过 return 语句返回发送的字符。putchar 函数的代码示例如下：

```

char putchar (char c)
{
    if (c == '\n')                // 扩展一行新字符
    {
        while (sendfull)          // 如果发送缓冲区满，则等待
        {

```

```
        out_task = os_running_task_id ();           //置位输出任务号
        os_wait (K_SIG, 0, 0);                     //等待信号
        out_task = 0xff;                           //输出任务号清零
    }
    putbuf (0x0D);                                  //在 LF 换行符之前发送 CR 回车符
}
while (sendfull)                                   //如果发送缓冲区满，则等待
{
    out_task = os_running_task_id ();               //置位输出任务号
    os_wait (K_SIG, 0, 0);                           //等待信号
    out_task = 0xff;                               //输出任务号清零
}
putbuf (c);                                         //发送字符
return c;                                           //返回字符
}
```

3. _getkey 函数

_getkey 函数用于获取串行口输入缓冲区的值。该函数中，通过 while 语句判断发送缓冲区 inbuf 的起点和终点是否一致，如果一致则等待，否则将通过 return 语句返回数值。_getkey 函数的代码示例如下：

```
char _getkey (void)
{
    while (inbuf_end == inbuf_start)                //等待
    {
        in_task = os_running_task_id ();            //置位输入任务号
        os_wait (K_SIG, 0, 0);                      //等待信号
        in_task = 0xff;                             //输入任务号清零
    }
    return (inbuf[inbuf_start++ & (INBUF_LEN-1)]); //返回值
}
```

4. 串行接收/发送中断处理函数

串行接收/发送中断处理函数 serial 用于处理串行口的中断，从而实现数据发送和接收。该中断处理函数使用工作寄存器组 2，程序中使用 if 语句来判断是接收中断还是发送中断。如果 RI 置 1，则表示接收中断，程序中对 switch 语句进行字符处理；如果 TI 置 1，则表示发送中断，首先清零中断请求标志 TI，然后通过 SBUF 发送字符。串行接收/发送中断处理函数 serial 的代码示例如下：

```
void serial () interrupt 4 using 2                  //中断响应函数
{
    unsigned char c;
    bit start_trans = 0;

    if (RI)                                         //RI 置 1，接收中断
    {
        c = SBUF;                                  //读 SBUF
        RI = 0;                                    //中断请求标志 RI 清零
        switch (c)                                  //字符处理
        {
            case CTRL_S:                            //如果是 CTRL_S
                sendstop = 1;                       //置位 sendstop，停止发送
        }
    }
}
```



```

        break;
    case CTRL_Q:                                //如果是 CTRL_Q
        start_trans = sendstop;                //开始发送
        sendstop = 0;
        break;
    default:                                    //对于其他字符
        if (inbuf_start + INBUF_LEN != inbuf_end) //则读入输入缓冲区
        {
            inbuf[inbuf_end++ & (INBUF_LEN-1)] = c;
        }

        if (in_task != 0xFF)                    //如果是任务等待
            isr_send_signal (in_task);          //发送信号
        break;
    }
}

if (TI || start_trans)                        //TI 置位, 发送中断
{
    TI = 0;                                    //中断请求标志 TI 清零
    if (outbuf_start != outbuf_end)            //如果输入缓冲区接收到字符
    {
        if (!sendstop)                        //发送字符
        {
            SBUF = outbuf[outbuf_start++ & (OUTBUF_LEN-1)];
            sendfull = 0;                      // sendfull 标志清零

            if (out_task != 0xFF)              //如果是任务等待
                isr_send_signal (out_task);    //发送信号
        }
    }
    else sendactive = 0;                       //全部发送完, sendactive 清零
}
}

```

5. 串行口初始化函数

串行口初始化函数 `serial_init` 用于初始化单片机的串行接口。该函数中, 首先初始化串行口为方式 1, 即 8 位 UART 方式。接着, 设置定时器的计数初值, 并启动定时器 T1, 允许串行口中断。串行口初始化函数 `serial_init` 的代码示例如下:

```

void init_serial (void)
{
    SCON = 0x50;                                //串行口方式 1, 8 位 UART, 允许接收
    TMOD = 0x20;                                //定时器方式 2, 8 位自动重装
    TH1 = 0xf3;                                  //波特率 2400bit/s
    TR1 = 1;                                     //启动定时器 T1
    ES = 1;                                      //允许串行口中断
}

```

32.3.4 获取命令函数

获取命令函数 `getline` 用来编辑从串口接收到的字符, 该函数位于 `GETCOMMAND.C` 文件中。这里通过 `_getkey` 函数获取串行口输入的命令, 然后对不同的命令进行不同的处理。程序规定了如下几个命令键及其代码。

51 单片机开发与应用技术详解

- CNTLQ，对应的字符代码为 0x11；
- CNTLS，对应的字符代码为 0x13；
- DEL，对应的字符代码为 0x7F；
- BACKSPACE，对应的字符代码为 0x08；
- CR，对应的字符代码为 0x0D；
- LF，对应的字符代码为 0x0A。

GETCOMMAND.C 文件及获取命令函数 getline 的代码示例如下：

```
#include <stdio.h>                                     //头文件

#define CNTLQ      0x11                                //定义控制字符
#define CNTLS      0x13
#define DEL        0x7F
#define BACKSPACE  0x08
#define CR         0x0D
#define LF         0x0A

void getline (char idata *line, unsigned char n)        //命令编辑函数
{
    unsigned char cnt = 0;
    char c;

    do                                                  //循环读取输入的命令
    {
        if ((c = _getkey ()) == CR)                    //读入字符
            c = LF;
        if (c == BACKSPACE || c == DEL)                //处理 BACKSPACE 键
        {
            if (cnt != 0)
            {
                cnt--;                                  //减 1
                line--;
                putchar (0x08);                          //回显 backspace 字符
                putchar (' ');
                putchar (0x08);
            }
        }
        else if (c != CNTLQ && c != CNTLS)              //忽略 CNTL_Q 字符和 NTL_S 字符
        {
            putchar (*line = c);                        //回显字符并保存
            line++;                                       //加 1
            cnt++;
        }
    } while (cnt < n - 1 && c != LF);                    //检查

    *line = 0;                                          //字符串结尾标志
}
```

当所有的程序输入完毕的时候，便可以进行程序的仿真和编译。如果程序通过编译，则可以将程序写入单片机中执行，便可以模拟道路交通灯的允许。

32.4 小结

本章详细讲述了道路交通灯的运行原理，以及如何使用 RTX-51 Tiny 程序来实现道路交通灯的控制模拟。本章给出了详细的电路图，以及 RTX-51 Tiny 的多任务程序。通过本章的学习，

可以掌握实时多任务操作系统的设计，尤其是基于 8051 单片机的 RTX-51 Tiny 的程序设计。

第 33 章 单总线温度传感器

DS18S20

单总线即 1-Wire 总线结构，是 Dallas Semiconductor 公司非常简单实用的总线协议。1-Wire 可以通过一条公共数据线实现主机与一个或多个从机之间的半双工、双向通信。使用 1-Wire 将引脚的使用减少到了最少，因此特别适合应用于单片机系统中。

Dallas Semiconductor 公司推出的 DS18S20 温度传感器即为 1-Wire 总线接口。由于其所需的引脚最少、接口简单、无须外部元件和精度高等优点，广泛应用于单片机系统中进行测温及温度监控。本章将主要介绍 1-Wire 总线及 1-Wire 总线接口的温度传感器 DS18S20 的应用。

33.1 单总线概述

单总线即 1-Wire 总线，顾名思义，是只需要一根数据线的数据传输方式。典型的 1-Wire 总线结构，如图 33-1 所示。其中，1-Wire 主机包括一个开漏极 I/O 端口，并通过上拉电阻上拉至 3.3V 或 5V 电源。外部 1-Wire 设备可以包含一个或多个，除了公共的地线外，所有 1-Wire 设备公用一根数据总线。1-Wire 总线结构中主机为数据传输的控制器，主动和 1-Wire 设备通信，而 1-Wire 设备只能被动和 1-Wire 主机通信。因此 1-Wire 总线结构是一种半双工的双向数据传输结构。

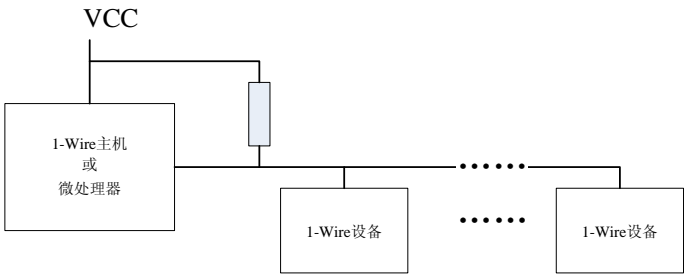


图 33-1 1-Wire 总线结构

所有的 1-Wire 设备在出厂时均有一个唯一的、不能更改的 64 位 ROM 序列号。这个序列号由激光刻制，永远不会与另一个器件重复。这个唯一的 ROM 序列号用于识别器件类型，以及从同一条纵向上的多个 1-Wire 设备中选择一个进行通信。

Wire 主机和 1-Wire 设备之间的通信格式，如图 33-2 所示。其中，SS 为 1-Wire 设备采样，MS 为 1-Wire 主机采样。1-Wire 总线的通信波形与脉宽调制类似，因为在数据位传输期间（或时隙）是通过宽脉冲（逻辑 0）和窄脉冲（逻辑 1）发送数据的。当 1-Wire 主机发出一个预定宽度的“复位”脉冲时，启动通信过程，并通过该脉冲同步整个总线系统。所有从机都会以一个逻辑低“应答”脉冲来响应复位脉冲。写数据时，1-Wire 主机首先拉低 1-Wire 总线以启动一个时隙，然后保持总线为低（宽脉冲）来发送逻辑 0，或释放总线（窄脉冲）使总线返回逻辑 1 状态。读数据时，1-Wire 主机以窄脉冲方式拉低总线，重新启动一个时隙；然后从机可以通过导通开漏极输出并保持线路为低来延长该脉冲，从而返回逻辑 0；或保持开漏极的关闭状

态以允许总线恢复，从而返回逻辑 1。

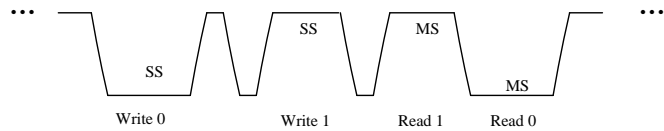


图 33-2 1-Wire 通信格式

大多数 1-Wire 器件都支持两种数据速率，即 15Kbps 标准速率和 111Kbps 高速速率。1-Wire 总线协议为自同步，并可接受数据位之间的较长延迟，从而确保了中断软件环境下的正常工作。

目前，市场上提供了多种 1-Wire 总线器件，包括 1-Wire 主机、1-Wire 存储器、1-Wire 温度传感器、1-Wire 可编码开关，以及 1-Wire 接口的 A/D 转换器等。使用 1-Wire 总线器件可以为系统带来多方面的好处，主要包括如下几点。

- 通过单线接口提供器件控制及操作，大大减少了 I/O 接口的使用；
- 每个 1-Wire 器件具有唯一的工厂光刻 ID，便于识别和选择器件；
- 可以选择使用“寄生电源”供电方式，而无需外加电源；
- 可以在单根数据线上挂接多个 1-Wire 器件；
- 1-Wire 器件提供了额外的 ESD 保护。

33.2 单总线温度传感器 DS18S20

DS18S20 是一种典型的 1-Wire 总线接口温度传感器，由 Dallas Semiconductor 公司生产。DS18S20 数字温度传感器提供了 9 位高精度的摄氏温度测量，同时具有非易失性、用户可编程上下触发门限的报警功能。由于其独特的 1-Wire 总线接口，使得其可以占用极少的 I/O 引脚资源，使用起来十分方便。

33.2.1 温度传感器 DS18S20 概述

DS18S20 采用带隙温度检测结构，是 DS1820 的升级产品。DS18S20 内部有三个主要部件，分别为 64 位激光刻制的唯一 ROM 序列号、温度传感器及非易失性温度报警触发器 TH 和 TL。DS18S20 通过 1-Wire 总线结构，仅需一个引脚即可实现数据的发送或接收。另外，用于 DS18S20 的供电电源可以从数据线本身获得，无须外部电源。每个 DS18S20 在出厂时都有唯一的一个 ROM 序列号，可以将多个 DS18S20 同时连在一根单总线上，从而实现多点分布温度测量。

DS18S20 以其简单方便的接口，广泛应用于温度测量、温度控制、数字温度计及热感测系统中。DS18S20 的主要特点如下：

- 1-Wire 单总线接口，通信仅需要一个 I/O 端口引脚；
- 每个器件具有唯一的、存储在片内 ROM 的 64 位序列码；
- 多节点检测功能简化了分布式温度检测应用；
- 使用简单方便，无须外部元件；
- 电源电压范围为 3.0V~5.5V，可选择由数据线供电；
- 可测量温度范围 -55℃~+125℃；
- 9 位数字温度计分辨率；
- 在 -10℃~+85℃ 温度范围内具有 ±0.5℃ 的高精度；
- 最大温度转换时间 750ms；
- 用户可编程的非易失性报警设置；
- 报警搜索命令能够自动识别和寻址温度超出设定门限（温度报警条件）之外的器件；
- 适合于包括温度测量、温度调节装置控制、工业系统、消费类产品、温度计及任何温度敏感系统的应用。

DS18S20 采用简单的 TO-92 封装，占用极少的电路板空间。DS18S20 的引脚排列，如图

33-3 所示。

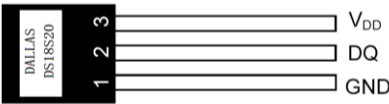


图 33-3 DS18S20 的引脚排列

DS18S20 各引脚功能如下所示。

- GND (Pin1): 接地引脚。
- DQ (Pin2): 1-Wire 总线的数据输入/输出引脚。
- V_{DD} (Pin3): 外部供电电源引脚。

33.2.2 DS18S20 的供电方式

DS18S20 可以采用两种供电方式，即外部供电方式和寄生电源供电方式。如果采用外部供电方式，如图 33-4 所示。此时 DS18S20 可以外接 3.3V 或 5V 的电源，而 GND 引脚必须接地。

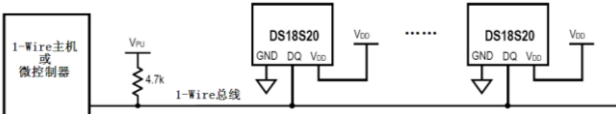


图 33-4 DS18S20 的外部供电方式

如果采用寄生电源供电方式，如图 33-5 所示。此时，DS18S20 的 V_{DD} 引脚必须接地。另外为了得到足够的工作电流，应给 1-Wire 线提供一个强上拉，一般可以使用一个场效应管将 I/O 线直接拉到电源上。DS18S20 从 1-Wire 单总线上汲取能量，在信号线 DQ 处于高电平期间把能量存储在内部电容里，在信号线 DQ 处于低电平期间消耗电容上的电量工作，直到高电平到来，再给 DS18S20 内部的寄生电源充电。

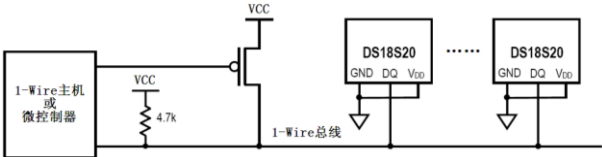


图 33-5 DS18S20 的寄生电源供电方式

在使用 DS18S20 时需要注意，如果温度高于 100℃，则不推荐使用寄生电源供电方式，而应采用外部电源供电方式。

33.2.3 DS18S20 的数据操作

1-Wire 总线将通信时使用的引脚减少到只有一根，在数据传输时需要满足特定的格式才能进行。1-Wire 总线通信的第一步是选择 1-Wire 设备，然后 1-Wire 主机发送各种命令来进行数据传输。

1. ROM 操作命令

1-Wire 总线协议选择 1-Wire 设备，主要是读取其内部的 64 位 ROM 序列号。在实际的通信过程中，1-Wire 主机通过如下的 5 个 ROM 操作命令来实现。

- 读出 ROM 序列号命令（代码为 33H），用于读出 DS18S20 的 64 位激光 ROM 序列号。
- 匹配 ROM 序列号命令（代码为 55H），用于识别（或选中）某一特定的 DS18S20 并进行后续操作。
- 搜索 ROM 序列号命令（代码为 F0H），用于确定 1-Wire 总线上的节点数，以及所有节点设备的 ROM 序列号。

- 跳过 ROM 序列号命令（代码为 CCH），用于等命令发出后，系统将对所有 DS18S20 进行操作，通常用于启动所有 DS18S20 进行温度转换之前，或 1-Wire 总线中仅有一个 DS18S20 时。
- 温度报警搜索命令（代码为 ECH），用于识别和定位系统中超出用户设定的报警温度界限的节点设备。

1-Wire 主机通过这些命令，对每个 DS18S20 的激光 ROM 部分进行操作。如果 1-Wire 总线上连接有多个器件，可以区分出每个 1-Wire 器件，同时可以向总线上的 1-Wire 主机报告有多少个 1-Wire 器件及 1-Wire 器件的类型。

2. 存储器操作命令

当通过 ROM 操作命令获取并选择特定的 1-Wire 从机后，1-Wire 主机便可以发出与该器件相关的操作命令，实现数据的读写。对于没有选定的 1-Wire 从机，均忽略该通信过程，直到 1-Wire 主机发出下一个复位脉冲。

DS18S20 内部存储器由一个高速暂存器和一个非易失性电可擦除 EEPROM 组成。DS18S20 存储器映像如图 33-6 所示。其中，高速暂存器用来保持数据的完整性，EEPROM 用来存储高低温报警触发值 TH 和 TL。

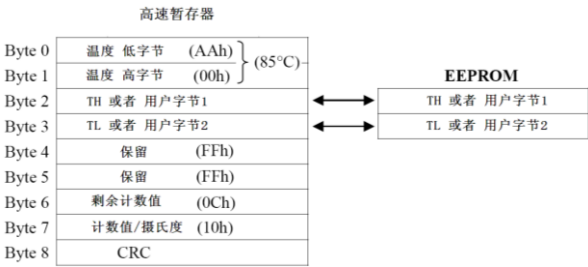


图 33-6 DS1820 的内部存储器映像

DS18S20 可以采用如下的存储器操作命令。

- 温度转换命令（代码为 44H），用于启动 DS18S20 进行温度测量。温度转换命令被执行后，DS18S20 进行温度测量和转换。如果使用外部电源供电，在 DS18S20 处于温度转换中，主机发送读时间隙，DS18S20 将在 1-Wire 总线上输出“0”，如果温度转换完成，则输出“1”。如果使用寄生电源供电，1-Wire 主机在发出温度转换命令后，必须立即启动强上拉并保持 750ms，在这段时间内 1-Wire 总线上不允许进行任何其他操作。
- 复制暂存器命令（代码为 48H），用于将高速暂存器中的内容复制到 DS18S20 的 EEPROM 中，即把温度报警器触发字节复制到非易失性存储器中。如果使用外部电源供电，DS18S20 在执行这条命令的过程中，主机发送读时间隙，DS18S20 将在 1-Wire 总线上输出一个“0”，如果复制过程结束的话，DS18S20 则输出“1”。如果使用寄生电源供电，1-Wire 主机必须在发出复制暂存器命令后，立即启动强上拉并最少保持 10ms，在这段时间内 1-Wire 总线上不允许进行任何其他操作。
- 写暂存器命令（代码为 4EH），用于将数据写入到 DS18S20 高速暂存器的地址 2（TH 字节）和地址 3（TL 字节）。当 DS18S20 执行写暂存器命令时，可以通过复位命令来中止写入。
- 读电源命令（代码为 B4H），用于读取 DS18S20 的供电方式。读电源命令执行后，通过读命令，将返回其供电模式，“0”表示使用寄生电源，“1”表示使用外部电源。
- 重读 EEPROM 命令（代码为 B8H），用于将存储在非易失性 EEPROM 中的内容重新读入到暂存器中。该命令在 DS18S20 上电时会自动执行，这样器件一开始工作，暂存器里便存在有效数据了。重读 EEPROM 命令执行后，如果执行读时间隙，DS18S20 会输出温度转换忙的标志。如果返回“0”表示忙，“1”表示温度转换完成。

- 读暂存器命令（代码为 BEH），用于读取高速暂存器中的内容。从高速暂存器字节 0 开始，最多读取 9 个字节。在读暂存器命令执行过程中，1-Wire 主机可以在任何时间发出复位命令来中止读取。

当 DS18S20 在 1-Wire 总线上通信时，高速暂存器用于确保数据的完整性。数据先被写入高速暂存器，并可被读回。数据经过 CRC 校验后，用一个复制暂存器命令将数据复制到非易失性 EEPROM 中。这一过程确保更改存储器时数据的完整性，从而提供了准确性。

通过 1-Wire 总线端口访问 DS18S20 的流程图，如图 33-7 所示。

DS18S20 需要严格的时序协议才能实现 1-Wire 总线通信。1-Wire 总线协议包括几种典型的信号类型，分别为复位脉冲、存在脉冲、写 0、写 1、读 0 和读 1。其中，存在脉冲由 1-Wire 从机发出，其余均由 1-Wire 主机发出。1-Wire 主机与 DS18S20 之间的任何操作都需要初始化开始。初始化时，1-Wire 主机发出复位脉冲，1-Wire 从机紧跟其后发出存在脉冲。存在脉冲通知 1-Wire 主机 DS18S20 在总线上已准备好，可以进行后续的 ROM 命令和存储器操作命令。

1-Wire 主机和 DS18S20 的直接数据读写是通过前面介绍的 ROM 操作命令、存储器操作命令及时间隙处理来实现的。时间隙包括写时间隙和读时间隙，对其介绍如下所示。

- 写时间隙：当 1-Wire 主机把数据线 DQ 从逻辑高电平拉到逻辑低电平的时候，写时间隙便开始。DS18S20 需要写 0 时间隙和写 1 时间隙两种写时间隙。当写时间隙开始后，DS18S20 在 $15\mu\text{s}\sim60\mu\text{s}$ 的时间窗口内对数据线 DQ 采样。如果 DQ 是低电平，就是写 0；否则，就是写 1。1-Wire 主机要发出一个写 1 时间隙，必须把数据线 DQ 拉到低电平然后释放，在写时间隙开始后的 $15\mu\text{s}$ 内，允许数据线 DQ 拉到高电平。1-Wire 主机要生成一个写 0 时间隙，必须把数据线拉到低电平并保持 $60\mu\text{s}$ 。
- 读时间隙：从 DS18S20 读取数据时，当 1-Wire 主机把数据线 DQ 从逻辑高电平拉到逻辑低电平时，读时间隙开始。数据线 DQ 必须至少持续 $1\mu\text{s}$ ；从 DS18S20 输出的数据在读时间隙的下降沿出现后 $15\mu\text{s}$ 内有效。此时，1-Wire 主机必须在这 $15\mu\text{s}$ 内停止把 DQ 引脚驱动为低电平，以读取数据线 DQ 状态。在读时间隙的结尾，数据线 DQ 将被外部上拉电阻拉到高电平。

从上面介绍可以看出，所有写时间隙必须至少持续 $60\mu\text{s}$ ，包括两个写周期及至少 $1\mu\text{s}$ 的总线恢复时间。所有读时间隙最少必须为 $60\mu\text{s}$ ，包括两个读周期和至少 $1\mu\text{s}$ 的恢复时间。

3. 温度转换操作

DS18S20 为 9 位数字温度分辨率，精度为 $0.5\text{ }^{\circ}\text{C}$ ，其温度数据格式如图 33-8 所示。DS18S20 的温度与数据对应关系如表 33-1 所示。所有数据都是以最低有效位（LSB）在前的方式进行读写的。

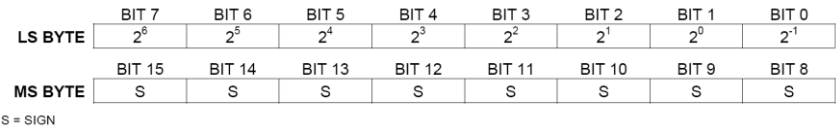


图 33-8 DS18S20 温度寄存器的数据格式

表 33-1 DS18S20 的温度与数据对应关系

温度 $^{\circ}\text{C}$	数据（二进制数）	数据（十六进制数）
+125 $^{\circ}\text{C}$	0000 0000 1111 1010	0x00FAH
+25 $^{\circ}\text{C}$	0000 0000 0011 0010	0x003H

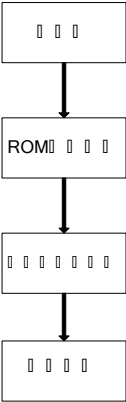


图 33-7 通过单总线访问 DS18S20 的流程图

+0.5℃	0000 0000 0000 0001	0x0001H
0℃	0000 0000 0000 0000	0x0000H
-0.5℃	1111 1111 1111 1111	0xFFFFH
-25℃	1111 1111 1100 1110	0xFFCEH
-55℃	1111 1111 1001 0010	0xFF92H

DS18S20 通过温度转换命令启动一次温度测量，测量结果存放在高速暂存器中，占有暂存器的字节 0（LSB）和字节 1（MSB）。由于 DS18S20 可以测量正负温度，因此测量数据是以 16 位带符号位扩展的二进制补码形式存放的。1-Wire 主机使用读暂存器命令可以把高速暂存器中的测量结果读出。

DS18S20 的温度报警触发器 TH 和 TL 各由一个 EEPROM 字节构成。1-Wire 主机对 TH 和 TL 的读取需要通过高速暂存器，而对 TH 和 TL 的写操作则直接使用写存储器命令即可。

虽然 DS18S20 的精度为±0.5℃，但是其提供了另外一种方法可以得到更高的精度。首先从高速暂存器读取字节 0 和字节 1 中的温度值，并去除最低有效位，即从读取的值中舍弃 0.5℃位，将该值记为“TEMP_READ”，然后读取高速暂存器的字节 6，记为“COUNT_REMAIN”，最后读取高速暂存器的字节 7，记为“COUNT_PER_C”。则扩展精度的温度值如下：

$$TEMPERATURE = TEMP_READ - 0.25 + \frac{COUNT_PER_C - COUNT_REMAIN}{COUNT_PER_C}$$

每完成一次温度转换后，DS18S20 自动将测量的温度值和温度报警限 TH 和 TL 中的值进行比较。如果温度值超出范围则置位其内部的报警标志。当报警标志被置位时，DS18S20 将会响应 1-Wire 主机的报警搜索命令。这样便可以实现多个 DS18S20 并联分布式测温。

33.3 单片机读写温度传感器 DS18S20 实例

一般的 51 系列单片机没有集成 1-Wire 总线控制器，因此常采用软件模拟的方法来实现。下面介绍如何使用 51 系列的单片机来读写 1-Wire 总线温度传感器 DS18S20。

33.3.1 电路图

这里采用 AT89S52 单片机来读写温度传感器 DS18S20。系统电路图如图 33-9 所示。该电路中所使用的元器件清单，如表 33-2 所示。

表 33-2 元器件列表

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89S52	1 个	R1	5kΩ	1 个
U2	DS18S20	1 个	R2	200Ω	1 个

续表

元 器 件	数 值	数 量	元 器 件	数 值	数 量
C1、C2	30pF	2 个	R3	10kΩ 排阻	1 个
C3	10μF 电解	1 个	R4	4.7kΩ	1 个
S1	按键开关	1 个	Y1	11.0592MHz	1 个

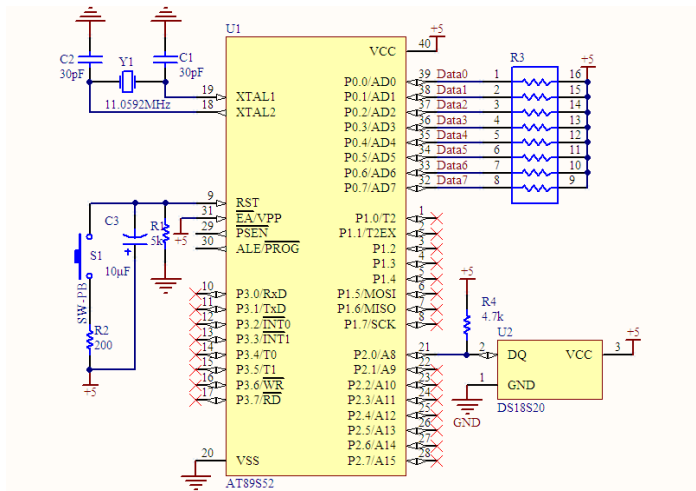


图 33-9 电路图

33.3.2 建立项目

- 首先在 Keil μ Vision3 集成开发环境中建立项目，具体操作步骤如下：
- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。
 - (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 33-10 所示。
 - (3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 33-11 所示。单击“是”按钮，完成工程的建立。
 - (4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

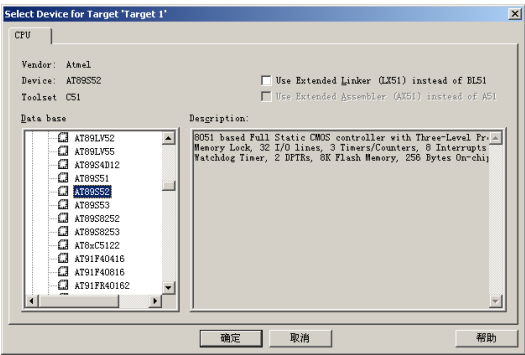


图 33-10 选择单片机 AT89S52

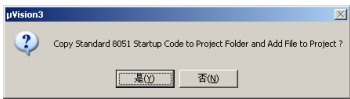


图 33-11 “ μ Vision3”对话框

33.3.3 DS18S20 读写子函数

对于 DS18S20 的操作需要严格遵守 1-Wire 总线协议。这里将 DS18S20 所支持的 ROM 操作命令、存储器操作命令等封装为子函数，方便调用。

1. 延时函数

延时函数 Delay 用于延时指定的时间，用来构成 1-Wire 总线协议所需要的时序。在程序中通过一个空循环语句便可以实现延时。延时函数 Delay 的程序代码示例如下：

```
void Delay(int useconds)
{
    int s;
```

```
for (s=0; s<useconds;s++);           //空循环语句实现延时
}
```

2. 复位函数

复位函数 **Reset** 用于完成 1-Wire 总线的复位操作。程序中首先将数据线 DQ 拉低并保持一段时间来实现 1-Wire 总线上所有器件的复位。接着主机等待 DS18S20 返回的存在脉冲，并返回存在信号。如果返回 0，则表示器件存在，返回 1，则表示无器件。复位函数 **Reset** 的程序代码示例如下：

```
uchar Reset(void)
{
    uchar PresenceSignal;
    DS18S20_DQ = 0;           //拉低数据线 DQ
    Delay(30);                 //延时
    DS18S20_DQ = 1;           //置数据线 DQ 为高电平
    Delay(3);                  //延时
    PresenceSignal = DS18S20_DQ; //读取存在信号
    Delay(30);                 //延时，等待时间隙结束
    return PresenceSignal;     //返回存在信号
}
```

3. 位写入函数

位写入函数 **WriteBit** 用于向 1-Wire 总线上的器件写入一位值。程序中首先拉低数据线 DQ 开始写时间隙，然后向 DQ 写入数据。如果写入 1 则数据线 DQ 置 1，如果写入 0 则数据线 DQ 置 0。位写入函数 **WriteBit** 的程序代码示例如下：

```
void WriteBit(char val)
{
    DS18S20_DQ = 0;           //拉低数据线 DQ 开始写时间隙
    if(val==1)
        DS18S20_DQ = 1;       //数据线 DQ 置 1，写 1
    else
        DS18S20_DQ=0;         //数据线 DQ 置 0，写 0
    Delay(5);                  //延时，在时间隙内保持电平值
    DS18S20_DQ = 1;           //拉高数据线 DQ
}
```

4. 字节写入函数

字节写入函数 **WriteByte** 用于向 1-Wire 总线上的器件写入一个字节数据。程序中采用循环移位的方式，每次调用位写入函数 **WriteBit** 写入一位。字节写入函数 **WriteByte** 的程序代码示例如下：

```
void WriteByte(char val)
{
    uchar i;
    uchar temp;
    for (i=0; i<8; i++)       //循环写入字节，每次写入一位
    {
        temp = val>>i;        //移位
        temp &= 0x01;
        WriteBit(temp);        //调用位写入函数
    }
    Delay(5);
}
```

5. 位读取函数

位读取函数 `ReadBit` 用于从 1-Wire 总线上读取从器件返回的一位值。程序中首先拉低数据线 DQ 开始读时间隙，然后将 DQ 置 1。然后延时一段时间，读取并返回数据总线 DQ 上的位数据。位读取函数 `ReadBit` 的程序代码示例如下：

```
uchar ReadBit(void)
{
    uchar i;
    DS18S20_DQ = 0;                //拉低数据总线 DQ 开始读时间隙
    DS18S20_DQ = 1;                //DQ 置 1
    for (i=0; i<3; i++);            //延时
    return DS18S20_DQ ;             //返回数据总线 DQ 上的位数据
}
```

6. 字节读取函数

字节读取函数 `ReadByte` 用于从 1-Wire 总线上读取从器件返回的一个字节数据。程序中采用循环移位的方式，每次调用位读取函数 `ReadBit` 读取一位。字节读取函数 `ReadByte` 的程序代码示例如下：

```
uchar ReadByte(void)
{
    uchar i;
    uchar value = 0;
    for (i=0; i<8; i++)             //读取字节，每次读取一位
    {
        if(ReadBit())
            value |= 0x01 << i;     //循环左移
        Delay(7);
    }
    return(value);                  //返回字节数据
}
```

7. 读取 ROM 代码函数

读取 ROM 代码函数 `ReadROMSerialNumber`，用于读取 1-Wire 总线上单个 DS18S20 器件的 ROM 代码。程序中首先使用 `Reset` 函数复位 1-Wire 总线上的所有器件，然后调用字节写入函数来执行读出 ROM 序列号命令(代码为 33H)。接着，循环调用字节读取函数来读取 DS18S20 返回的 8 个 ROM 序列号字节。最后，通过串口输出该 ROM 序列号。读取 ROM 代码函数 `ReadROMSerialNumber` 的程序代码示例如下：

```
void ReadROMSerialNumber(void)
{
    int n;
    char dat[9];
    printf("\nReading DS18S20 ROM Code\n");    //输出信息
    Reset();                                    //复位函数
    WriteByte(0x33);                            //读出 ROM 序列号命令
    for (n=0; n<8; n++)
    {
        dat[n]=ReadByte();                      //循环读 ROM 序列号
    }
    printf("\nDS18S20 ROMCode=%X%X%X%X\n",      //输出 ROM 序列号
        dat[7], dat[6], dat[5], dat[4], dat[3], dat[2], dat[1], dat[0]);
}
```

8. CRC 校验函数

CRC 校验函数 CRCCheck 用于完成一次循环冗余校验。一般来说，在进行 ROM 搜索时需要 CRC 校验。CRC 校验函数 CRCCheck 的程序代码示例如下：

```
uchar CRCCheck( uchar x)
{
    CRCdsc = dsc[CRCDsc^x];           //查表校验
    return CRCdsc;
}
```

其中采用了查表法来实现 CRC 校验，查表法的 CRC 校验表，其详细介绍可参阅 DS18S20 的参考应用笔记。在程序设计时，需要预先定义该 CRC 校验表，示例如下：

```
uchar code dsc[] =
{
    0, 94,188,226, 97, 63,221,131,194,156,126, 32,163,253, 31, 65,
    157,195, 33,127,252,162, 64, 30, 95, 1,227,189, 62, 96,130,220,
    35,125,159,193, 66, 28,254,160,225,191, 93, 3,128,222, 60, 98,
    190,224, 2, 92,223,129, 99, 61,124, 34,192,158, 29, 67,161,255,
    70, 24,250,164, 39,121,155,197,132,218, 56,102,229,187, 89, 7,
    219,133,103, 57,186,228, 6, 88, 25, 71,165,251,120, 38,196,154,
    101, 59,217,135, 4, 90,184,230,167,249, 27, 69,198,152,122, 36,
    248,166, 68, 26,153,199, 37,123, 58,100,134,216, 91, 5,231,185,
    140,210, 48,110,237,179, 81, 15, 78, 16,242,172, 47,113,147,205,
    17, 79,173,243,112, 46,204,146,211,141,111, 49,178,236, 14, 80,
    175,241, 19, 77,206,144,114, 44,109, 51,209,143, 12, 82,176,238,
    50,108,142,208, 83, 13,239,177,240,174, 76, 18,145,207, 45,115,
    202,148,118, 40,171,245, 23, 73, 8, 86,180,234,105, 55,213,139,
    87, 9,235,181, 54,104,138,212,149,203, 41,119,244,170, 72, 22,
    233,183, 85, 11,136,214, 52,106, 43,117,151,201, 74, 20,246,168,
    116, 42,200,150, 21, 75,169,247,182,232, 10, 84,215,137,107, 53
};
```

9. 搜索器件函数

搜索器件函数 SearchDevice 用于搜索 1-Wire 总线上的下一个 DS18S20 器件。在程序中主机循环搜索，直到全部 ROM 字节 0~7 都完成。如果 1-Wire 总线上没有其他 DS18S20 器件则返回“FALSE”。在 ROM 搜索时，主机写入搜索 ROM 序列号命令（代码为 F0H）并需要执行 CRC 校验。搜索器件函数 SearchDevice 的程序代码示例如下：

```
uchar SearchDevice(void)
{
    uchar m = 1;           //DS18S20 ROM 位索引
    uchar n = 0;           //DS18S20 ROM 字节索引
    uchar k = 1;
    uchar x = 0;
    uchar discrepMarker = 0;
    uchar g;
    uchar nxt;
    int flag;
    nxt = FALSE;
    CRCDsc = 0;
    flag = Reset();         //复位函数
    if(flag||EndFlag)       //如果没有其他器件
    {
        LastData = 0;
        return FALSE;      //返回“FALSE”
    }
    WriteByte(0xF0);        //搜索 ROM 命令
    do                      //循环
    {
```

```

x = 0;
if(ReadBit()==1) x = 2;
Delay(8);
if(ReadBit()==1) x |= 1;
if(x ==3) break;
else
{
    if(x>0)
        g = x>>1;
    else
    {
        if(m<LastData)
            g = ((DS18S20ROM[n]&k)>0);
        else
            g = (m==LastData);
        if (g==0) discrepMarker = m;
    }
    if(g==1)
        DS18S20ROM[n] |= k;
    else
        DS18S20ROM[n] &= ~k;
    WriteBit(g); //位写入函数
    m++;
    k = k<<1;
    if(k==0)
    {
        CRCCheck(DS18S20ROM[n]); //CRC 校验
        n++; k++;
    }
}
}while(n<8); //直到全部 ROM 字节 0~7 都完成
if(m<65||CRCdsc)
    LastData=0;
else //搜索成功
{
    LastData = discrepMarker; //置位 LastData, lastOne 和 nxt
    EndFlag = (LastData==0);
    nxt = TRUE; //表示总线上还有其他器件，搜索未结束
}
return nxt;
}

```

10. 搜索第一个器件函数

搜索第一个器件函数 `FindFirstDevice` 用于搜索 1-Wire 总线上的第一个 DS18S20。在程序中,主要调用搜索器件函数 `SearchDevice` 来完成一次搜索。搜索第一个器件函数 `FindFirstDevice` 的程序代码示例如下:

```

uchar FindFirstDevice(void)
{
    LastData = 0;
    EndFlag = FALSE;
    return SearchDevice(); //搜索器件函数 SearchDevice
}

```

11. 读取暂存器函数

读取暂存器函数 `ReadData` 用于读取 DS18S20 内部高速暂存器。程序中首先执行读暂存器命令(代码为 BEH),然后通过循环调用字节读取函数来读取暂存器中 9 个字节的数据。读取暂存器函数 `ReadData` 的程序代码示例如下:

```

void ReadData(void)
{

```

51 单片机开发与应用技术详解

```
int j;
char pad[10];
printf("\nReading ScratchPad Data\n");
WriteByte(0xBE); //读暂存器命令 ( 代码为 BEH )
for (j=0;j<9;j++) //循环读取暂存器中 9 个字节的数据
{
    pad[j]=ReadByte(); //字节读取函数
}
printf("\n ScratchPAD DATA =%X%X%X%X%X\n",
pad[8],pad[7],pad[6],pad[5],pad[4],pad[3],pad[2],pad[1],pad[0]);
}
```

12. 查找器件函数

查找器件函数 FindDevices 用于查找 1-Wire 总线上的所有 DS18S20 器件。程序中首先复位 1-Wire 总线，通过返回值确定是否存在任何器件。如果总线上存在 DS18S20，则将其唤醒并调用。程序中使用 FindFirstDevice 函数查找第一个器件，并返回到 SearchDevice 函数。SearchDevice 函数用于识别总线上每个器件唯一 ROM 序列号。查找器件函数 FindDevices 的程序代码示例如下：

```
void FindDevices(void)
{
    uchar m;
    if(!Reset()) //复位总线
    {
        //如果存在器件则开始处理
        if(FindFirstDevice()) //调用 FindFirstDevice 函数
        {
            numROMs=0;
            do //循环
            {
                numROMs++;
                for (m=0;m<8;m++)
                {
                    //识别 ROM 代码
                    ROMFound[numROMs][m]=DS18S20ROM[m];
                }
                printf("\nDS18S20 ROM CODE =%02X%02X%02X%02X\n",
                ROMFound[5][7],ROMFound[5][6],ROMFound[5][5],ROMFound[5][4],
                ROMFound[5][3],ROMFound[5][2],ROMFound[5][1],ROMFound[5][0]);
            }while (SearchDevice() && (numROMs<10)); //直到没有发现其他器件
        }
    }
}
```

13. 读取温度函数

读取温度函数 ReadTemperature 用于读取 DS18S20 测量的温度。如果 1-Wire 总线上只有一个 DS18S20 可以使用该函数来获取测量温度。程序中首先复位 1-Wire 总线，然后启动温度转换命令（代码为 44H）。接着通过读暂存器命令（代码为 BEH）读取温度数据，最后通过处理输出对应的摄氏温度值及华氏温度值。读取温度函数 ReadTemperature 的程序代码示例如下：

```
void ReadTemperature(void)
{
    char get[10];
    char temp_lsb,temp_msb;
    int k;
    char Ftemperature,Ctemperature;
    Reset(); //复位
    WriteByte(0xCC); //跳过 ROM 序列号命令 ( 代码为 CCH )
    WriteByte(0x44); //启动温度转换命令 ( 代码为 44H )
}
```

```

Delay(5);
Reset();
WriteByte(0xCC);           //跳过 ROM 序列号命令 ( 代码为 CCH )
WriteByte(0xBE);           //读暂存器命令 ( 代码为 BEH )
for (k=0;k<9;k++)
{
    get[k]=ReadByte();      //循环读取
}
printf("\n Scratch DATA = %X%X%X%X\n",
get[8],get[7],get[6],get[5],get[4],get[3],get[2],get[1],get[0]);
temp_msb = get[1];
temp_lsb = get[0];
if (temp_msb <= 0x80)
{
    temp_lsb = (temp_lsb/2);    //移位, 得到完整的温度值
}
temp_msb = temp_msb & 0x80;    //屏蔽符号位之外的所有数据位
if (temp_msb >= 0x80)
{
    temp_lsb = (~temp_lsb)+1;   // temp_lsb 取补
}
if (temp_msb >= 0x80)
{
    temp_lsb = (temp_lsb/2);    //移位, 得到完整的温度值
}
if (temp_msb >= 0x80)
{
    temp_lsb = ((-1)*temp_lsb); //符号位
}
printf( "\nTempC= %d degrees C\n", (int)temp_lsb );    //摄氏温度值输出
Ctemperature = temp_lsb;
Ftemperature = (((int) Ctemperature)* 9)/5 + 32;
printf( "\nTempF= %d degrees F\n", (int)Ftemperature ); //华氏温度值输出
}

```

33.3.4 主函数

这里通过 AT89S52 来作为 1-Wire 的主机, 定义 P2.0 引脚为 1-Wire 的数据总线。主函数中首先初始化串行口为模式 1, 波特率为 4800bit/s。接着通过 while 循环语句来扫描串口输入, 根据输入数据来调用函数执行不同的功能。主函数的程序代码示例如下:

```

#include <stdio.h>           //头文件
#include <reg52.h>

#define FALSE 0
#define TRUE 1
#define uchar unsigned char //预定义

sbit DS18S20_DQ = P2^0;     //定义 P2^0 引脚为 DS18S20 的 DQ 引脚

/***** 全局变量声明 *****/
uchar DS18S20ROM[8];        // DS18S20 ROM 位
uchar LastData = 0;
uchar EndFlag = 0;
uchar ROMFound[5][8];       // DS18S20 的 ROM 代码表
uchar numROMs;
uchar CRCdsc;               //用于 CRC 校验

```


51 单片机开发与应用技术详解

```
void main(void)
{
    uchar Select_Menu;                                //功能选择

    SCON=0x50;                                        //初始化串行口模式 1
    TMOD=0x20;                                        //初始化 T1 为定时功能，模式 2
    PCON=0x80;                                        //设置 SMOD=1
    TL1=0xF4;                                        //波特率 4800bit/s，初值
    TH1=0xF4;

    TR1 = 1;                                          //启动 T1
    TI = 1;                                          //启动发送

    while(1)                                          //主循环
    {
        printf (" AT89S52 Control DS18S20\n");      //输出信息
        printf("\n*****\n");
        printf (" Select Control Menu Option\n");
        printf (" 1. Reset 1-Wire.\n");
        printf (" 2. Read DS18S20 ROM Code.\n");
        printf (" 3. Search DS18S20 ROM.\n");
        printf (" 4. Find All DS18S20.\n");
        printf (" 5. Read DS18S20 Scratch.\n");
        printf (" 6. Read DS18S20 Temperature.\n");
        printf (" 7. Exit.\n");
        printf (" Please Input Menu Option:\n");

        Select_Menu = _getkey();                      //从键盘输入选择数字
        switch(Select_Menu)
        {
            case '1':                                //复位 1-Wire 总线
                printf ("\n You Select 1. Reset 1-Wire\n");
                Reset();                               //复位函数
                break;

            case '2':                                //读 DS18S20 ROM
                printf ("\n You Select 2. Read DS18S20 ROM Code \n");
                Reset();
                ReadROMSerialNumber();                 //读取 ROM 代码函数
                break;

            case '3':                                //搜索 DS18S20
                printf("\n You Select 3. Search DS18S20 ROM \n");
                Reset();
                FindFirstDevice();                     //搜索第一个器件
                printf("\nDS18S20 ROM CODE =%02X%02X%02X\n",
                    ROMFound[5][7],ROMFound[5][6],ROMFound[5][5],ROMFound[5][4],
                    ROMFound[5][3],ROMFound[5][2],ROMFound[5][1],ROMFound[5][0]);
                break;

            case '4':                                //搜索所有 DS18S20
                printf ("\n You Select 4. Find All DS18S20\n");
                Reset();
                FindDevices();                          //查找器件函数
                break;

            case '5':                                //读取高速暂存器
                printf ("\n You Select 5. Read DS18S20 Scratch \n");
                Reset();
```

```

WriteByte(0xCC);                //跳过 ROM 序列号命令 ( 代码为 CCH )
ReadData();                    //读取高速暂存器
break;

case '6':                        //读取温度
printf ("\n You Select 6. Read DS18S20 Temperature\n");
ReadTemperature();              //读取温度值
break;

case '7':                        //退出程序
printf ("\n You Select 7. Exit\n");
goto Exit;                      //转向 Exit 标号处
break;

default:
printf ("\n Error: Please Select Right Menu Option\n");
break;
};

}

Exit: printf("Exit the program!"); //退出
}

```

通过前面的 DS18S20 读写子函数及 main 主函数，便可以实现 AT89S52 控制 1-Wire 总线温度传感器 DS18S20。

33.3.5 程序仿真

Keil μ Vision3 集成开发环境提供了很好的信号仿真功能，下面就利用其进行程序的仿真分析。具体操作步骤如下：

- (1) 在 Keil μ Vision3 集成开发环境中，选择“Debug”→“Start/Stop Debug Session”命令，进入仿真分析模式。
- (2) 选择“Peripherals”→“Serial”命令，打开“串口仿真”接口。
- (3) 选择“Debug”→“Go”命令，开始执行仿真。
- (4) 此时，在串行仿真接口中便输出 DS18S20 的控制菜单。这里输入“6”，即选择读取温度值。程序便输出对应的摄氏温度值及华氏温度值，如图 33-12 所示。



图 33-12 仿真结果

当仿真无误后，便可以将程序下载到单片机中执行。

33.4 小结

本章介绍了 1-Wire 单总线的工作原理，并结合 1-Wire 总线接口温度传感器 DS18S20，详细讲解了其供电方式及数据操作命令。最后通过一个完整的实例介绍了如何使用 51 系列单片机模拟 1-Wire 总线数据传输，从而实现 DS18S20 的控制。1-Wire 单总线是一种结构简单的接

51 单片机开发与应用技术详解

口协议，其最大化地减少了 I/O 引脚数目，在实际电路中有着广泛的应用。

第 34 章 Microware 串行总线

EEPROM 的应用

Microware 同步串行总线接口是 National Semiconductor 公司提出的串行同步双工通信接口。由于其采用三根信号线，所以常称为三线制同步串行总线接口。三线制 Microware 同步串行总线接口最早应用在 National Semiconductor 公司的 COP 系统和 HPC 系列微控制器上，后来被广泛应用到其他微控制器、存储器、A/D 转换器和 D/A 转换器等领域。

三线制 Microware 串行总线占用 I/O 引脚少，使用简单方便，可以提高系统的可靠性。本章将主要介绍三线制 Microware 串行总线接口，以及采用 Microware 串行总线接口的 EEPROM 存储器的使用。

34.1 三线制 Microware 串行总线概述

三线制 Microware 同步串行总线接口是一种串行同步双工通信接口，由美国 National Semiconductor 公司最早提出。Microware 串行总线使用如下所述的三根信号线进行通信。

- SK，时钟信号线；
- SI，数据输入线；
- SO，数据输出线。

典型的 Microware 串行总线系统包括 Microware 主机及多个 Microware 从机。Microware 主机的数据输出线 SO 和所有从机的数据输入线相接，从机的数据输出线都接到主机的数据输入线 SI 上。所有从机的时钟线连接到同一根 SK 线上。工作时，Microware 主机向 SK 线发送时钟脉冲信号，从机在时钟信号 SK 的同步沿输出/输入数据。

对于包含多个 Microware 从机的系统，每个从机都需要提供一根片选通信信号线 CS。当 CS 为高电平的时候，选通对应的 Microware 从机进行通信。多个 Microware 从机的系统图，如图 34-1 所示。在 Microware 串行总线数据传输过程中，数据交换采用高位在前的格式。

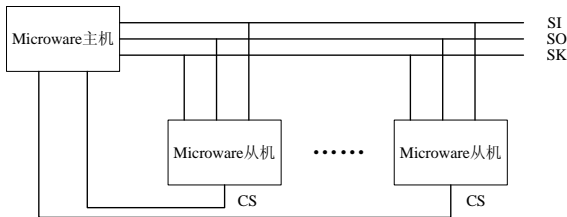


图 34-1 多个 Microware 从机的系统图

三线制 Microware 串行总线最少只需三根数据线和一根地线。相比并行总线接口，其减少了 I/O 引脚的使用，可以提高系统的可靠性。因此，现在很多种器件均提供了 Microware 串行总线接口。

34.2 Microware 串行总线接口的 EEPROM

由于三线制 Microware 串行总线接口的优势，目前多家公司推出了各种 Microware 串行总线接口的产品。其中，使用最多的便是 Microware 串行总线接口的 EEPROM 存储器。

34.2.1 Microware 串行总线接口 EEPROM 概述

EEPROM 是一种可用电气方法在线擦除和写入的存储器。EEPROM 既有普通 RAM 存储器可读可写的特性，又具有非易失性存储器 ROM 在掉电后仍然能保持所存储数据的优点。许多大的半导体公司都生产可支持三线制 Microware 串行总线的 EEPROM。例如，National Semiconductor 公司推出的 NM93C06/46/56/66 产品，Microchip 公司推出的 93C06/46/56/66、93LC46/56/66 和 93AA06/46/56，以及 Atmel 公司的 AT93C46/56/57/66 等产品。这些三线制 Microware 串行总线的 EEPROM 都具有如下特点。

- 一般采用单一+5V 电源供电；
- 具有较低的功耗；
- 具有三态输出，并可与 TTL 电平兼容；
- 擦除/写入时间不超过 10ms；
- 擦除/写入周期寿命一般都可达到 10 万次以上，有的产品已可达到 100 万次；
- 片内写入的数据保存寿命可达 40 年以上。

下面以 Atmel 公司的 AT93C46/56/57/66 系列为例，介绍三线制 Microware 串行总线 EEPROM。AT93C46/56/57/66 分别具有 1Kbit、2Kbit 和 4Kbit 的存储空间，其引脚排列如图 34-2 所示。

AT93C46/56/57/66 芯片各引脚功能介绍如下：

- CS：片选信号；
- SK：Microware 串行时钟输入信号，数据在其上升沿锁定有效；
- DI：Microware 数据输入引脚；
- DO：Microware 数据输出引脚；
- GND：接地引脚；
- ORG：内部存储器组织引脚；当 ORG 接 VCC 时，其内部存储组织结构以 16 位为一个单元；当 ORG 接 GND 时，其内部存储组织结构以 8 位为一个单元；
- DC：空引脚，不连接；
- VCC：接+5V 电源。

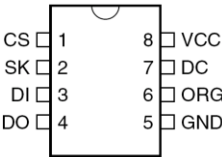


图 34-2 AT93C46/56/57/66 的引脚排列

34.2.2 Microware 串行总线接口 EEPROM 的指令

Atmel 公司的 AT93C46/56/57/66 系列 Microware 串行 EEPROM 包含 7 个指令，这些指令和 Microware 串行总线兼容。其中，对于不同的型号，其具体的指令所代表的含义略有区别。下面分别进行介绍。

AT93C46 具有 1Kbit 的存储空间，可以按照 128×8 或 64×16 来使用。AT93C46 所支持的 Microware 串行总线指令及其说明，如表 34-1 所示。

表 34-1 AT93C46 的指令集

指 令	起始位	操作码	地 址		数 据		说 明
			×8	×16	×8	×16	
READ	1	10	A6~A0	A5~A0			读取指定地址存储单元的数据
EWEN	1	00	11×××××	11××××			写使能

续表

指 令	起始位	操作码	地 址		数 据		说 明
			×8	×16	×8	×16	

ERASE	1	11	A6~A0	A5~A0			擦除指定存储单元的数据
WRITE	1	01	A6~A0	A5~A0	D7~D0	D15~D0	在指定的存储单元写入数据
ERAL	1	00	10xxxxxx	10xxxx			擦除所有存储单元
WRAL	1	00	01xxxxxx	01xxxx	D7~D0	D15~D0	写所有存储单元
EWDS	1	00	00xxxxxx	00xxxx			禁止一切与编程有关的操作

AT93C57 具有 2Kbit 的存储空间，可以按照 256×8 或 128×16 来使用。AT93C57 所支持的 Microware 串行总线指令及其说明，如表 34-2 所示。

表 34-2 AT93C57 的指令集

指 令	起始位	操作码	地 址		数 据		说 明
			×8	×16	×8	×16	
READ	1	10	A7~A0	A6~A0			读取指定地址存储单元的数据
EWEN	1	00	11xxxxxx	11xxxx			写使能
ERASE	1	11	A7~A0	A6~A0			擦除指定存储单元的数据
WRITE	1	01	A7~A0	A6~A0	D7~D0	D15~D0	在指定的存储单元写入数据
ERAL	1	00	10xxxxxx	10xxxx			擦除所有存储单元
WRAL	1	00	01xxxxxx	01xxxx	D7~D0	D15~D0	写所有存储单元
EWDS	1	00	00xxxxxx	00xxxx			禁止一切与编程有关的操作

AT93C56/66 具有 4Kbit 的存储空间，可以按照 512×8 或 256×16 来使用。AT93C56/66 所支持的 Microware 串行总线指令及其说明，如表 34-3 所示。

表 34-3 AT93C56/66 的指令集

指 令	起始位	操作码	地 址		数 据		说 明
			×8	×16	×8	×16	
READ	1	10	A8~A0	A7~A0			读取指定地址存储单元的数据
EWEN	1	00	11xxxxxx	11xxxx			写使能
ERASE	1	11	A8~A0	A7~A0			擦除指定存储单元的数据
WRITE	1	01	A8~A0	A7~A0	D7~D0	D15~D0	在指定的存储单元写入数据
ERAL	1	00	10xxxxxx	10xxxx			擦除所有存储单元
WRAL	1	00	01xxxxxx	01xxxx	D7~D0	D15~D0	写所有存储单元
EWDS	1	00	00xxxxxx	00xxxx			禁止一切与编程有关的操作

34.2.3 Microware 串行总线接口 EEPROM 的指令时序

前面介绍了 AT93C46/56/57/66 系列 EEPROM 所支持的 Microware 串行指令。下面分别介绍各个指令执行的时序。

1. READ 指令

READ 指令用于从指定的地址中读出数据。当片选信号 CS 为高电平时，当 DI 引脚输入 READ 指令码及待读取的地址码后，对应地址上的数据便通过 DO 引脚串行输出。数据在串行时钟 SK 的上升沿有效，并且高位数据先输出，低位数据最后输出。READ 指令时序图，如图 34-3 所示。

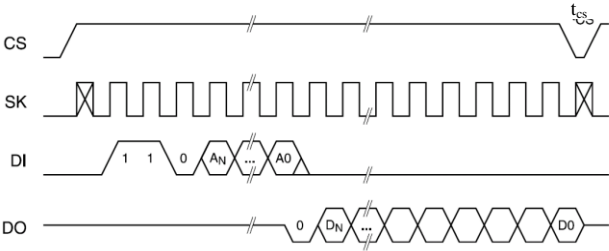


图 34-3 READ 指令时序图

2. EWEN 指令

EWEN 指令用于 EEPROM 的写使能。为了保证数据的完整性，三线制 Microware 串行 EEPROM 在上电的时候，便自动进入擦除/写禁止状态。因此，在写 EEPROM 或执行擦除命令时，首先需要执行 EWEN 命令来使 EEPROM 进入擦除/写使能状态。EWEN 指令时序图，如图 34-4 所示。

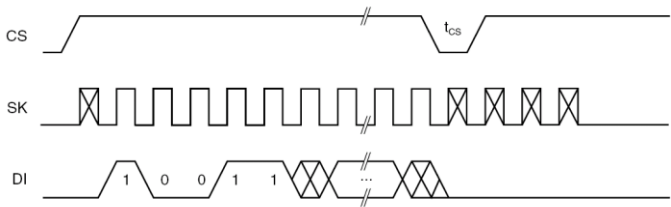


图 34-4 EWEN 指令时序图

3. ERASE 指令

ERASE 指令用于擦除指定的存储区，并将其逻辑状态变为 1。当 ERASE 指令执行后，可以通过 DO 引脚来判断操作是否结束。DO 引脚的高电平表示选定的存储区已经被擦除，下面可以执行其他指令。ERASE 指令时序图，如图 34-5 所示。

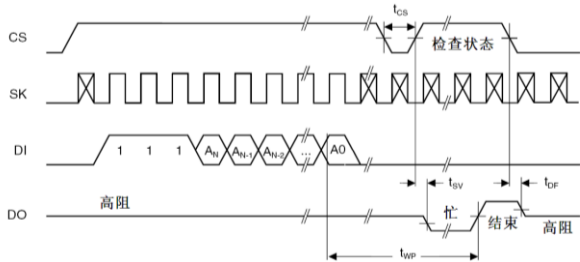


图 34-5 ERASE 指令时序图

4. WRITE 指令

WRITE 指令用于将数据写入指定的存储区。当 WRITE 指令执行后，可以通过 DO 引脚来判断操作是否结束。DO 引脚的低电平表示还在执行当前指令；DO 引脚的高电平表示数据写操作已经结束，下面可以执行其他指令。WRITE 指令时序图，如图 34-6 所示。

5. ERAL 指令

ERAL 指令用于擦除所有的存储区，将其逻辑状态变为 1。当 ERAL 指令执行后，可以通过 DO 引脚来判断操作是否结束。DO 引脚的高电平表示数据写操作已经结束，下面可以执行其他指令。ERAL 指令时序图，如图 34-7 所示。

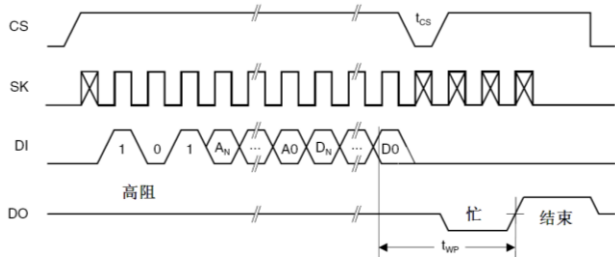


图 34-6 WRITE 指令时序图

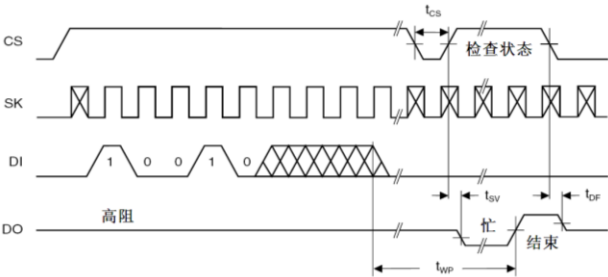


图 34-7 ERAL 指令时序图

6. WRAL 指令

WRAL 指令用于写所有的存储区。当 WRAL 指令执行后，可以通过 DO 引脚来判断操作是否结束。DO 引脚的低电平表示还在执行当前指令；DO 引脚的高电平表示数据写操作已经结束，下面可以执行其他指令。WRAL 指令时序图，如图 34-8 所示。

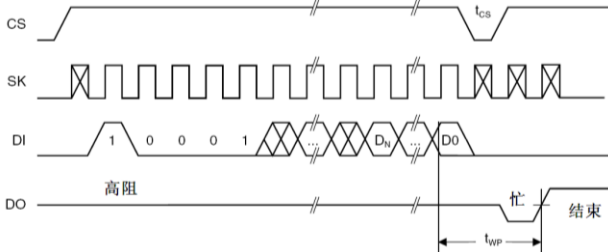


图 34-8 WRAL 指令时序图

7. EWDS 指令

EWDS 指令用于保护数据，将 EEPROM 置为擦除/写禁止状态。一般来说，在所有写操作和擦除操作完成后再执行 EWDS 指令。当 EEPROM 置于擦除/写禁止状态时，READ 指令仍然可以使用。EWDS 指令时序图，如图 34-9 所示。

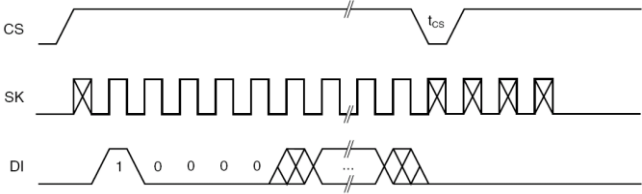


图 34-9 EWDS 指令时序图

通过这些指令及其时序，便可以在没有三线制 Microware 串行总线接口的微处理器上，使用软件模拟的方式来实现 Microware 串行总线控制。

34.3 51 系列单片机读写三线制 EEPROM 实例

在基本的 51 系列单片机中，没有集成三线制 Microware 串行总线。当其与三线制 Microware 串行总线的 EEPROM 连接时，只能够采用软件模拟的方式来实现三线制 Microware 主机的操作。下面以 Atmel 公司的 AT89S52 单片机及 AT93C66 三线制 Microware 串行 EEPROM 为例，介绍如何在软件上实现三线制 Microware 串行的数据操作。

34.3.1 电路图

这里采用 AT89S52 单片机来读写三线制 Microware 串行 EEPROM 芯片 AT93C66。系统电路图如图 34-10 所示。该电路中所使用的元器件清单，如表 34-4 所示。

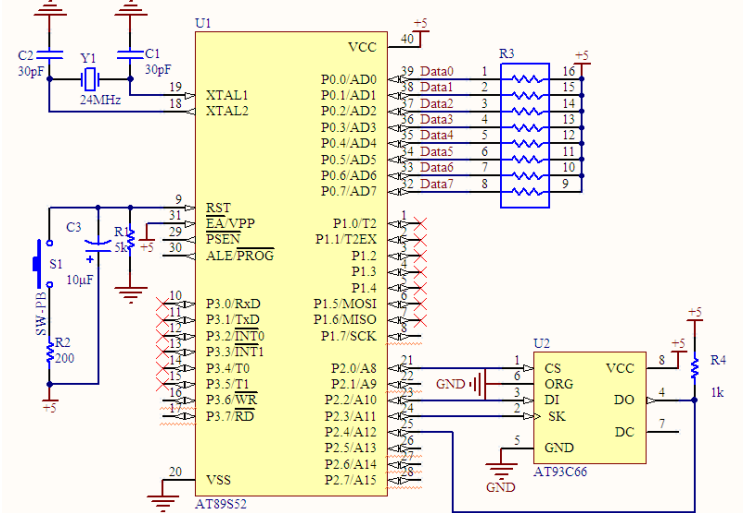


图 34-10 电路原理图

表 34-4 元器件列表

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89S52	1 个	R1	5k Ω	1 个
U2	AT93C66	1 个	R2	200 Ω	1 个
C1、C2	30pF	2 个	R3	10k Ω 排阻	1 个
C3	10 μ F 电解	1 个	R4	1k Ω	1 个
S1	按键开关	1 个	Y1	11.0592MHz	1 个

这里使用 AT89S52 的 P2.0 引脚连接 AT93C66 的片选引脚 CS。AT93C66 的 SK 连接 P2.3 引脚，DI 连接 P2.2 引脚，DO 通过上拉电阻连接 P2.4 引脚。AT93C66 的存储器组织引脚 ORG 直接接地，使用 8 位存储器结构。

34.3.2 建立项目

首先在 Keil μ Vision3 集成开发环境中建立项目，具体操作步骤如下：

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 34-11 所示。
- (3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 34-12 所示。单击“是”按钮，完成工程的建立。

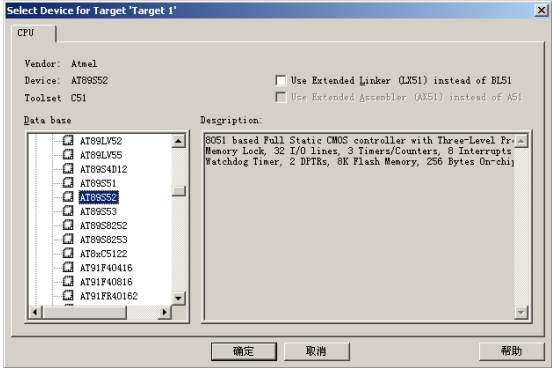


图 34-11 选择单片机 AT89S52



图 34-12 “u Vision3”对话框

(4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

34.3.3 三线制 Microware 串行总线读写子函数

对于三线制 Microware 串行总线的操作需要严格遵守总线协议。这里将 AT93C66 所用到的操作指令封装为子函数，方便调用。这些子函数不仅适用于三线制 Microware 串行 EEPROM，同样适用于其他 Microware 串行接口的器件。下面分别进行介绍。

1. 延时函数

延时函数 Delay 用于产生规定的软件延时，以实现 Microware 的时序要求。在程序中，使用一个空循环函数来实现程序延时。延时函数 Delay 的程序代码示例如下：

```
void Delay(uchar n)
{
    int i;
    for(i=0;i<n;i++);           //空延时
}
```

2. 时钟函数

时钟函数 MicrowareClock 用于在 SK 信号线上产生时钟信号，供数据传输同步使用。程序中通过分别置信号线 SK 为高电平和低电平来实现时钟信号。时钟函数 MicrowareClock 的程序代码示例如下：

```
void MicrowareClock(void)
{
    SK=Low;                       //SK=0
    Delay(2);
    SK=High;                      //SK=1
    Delay(2);
}
```

3. 字节输入函数

字节输入函数 MicrowareSend 用于向 Microware 总线上的从器件发送一个字节的数。程序中使用 for 循环语句来循环移位逐个输出字节数据。其中，字节的高位首先输出。字节输入函数 MicrowareSend 的程序代码示例如下：

```
void MicrowareSend(uchar senddata)
{
    int i;
    for(i=0;i<8;i++)             //循环输出一个字节的数
    {
        DI=(bit)(senddata>>80);
```

```

        senddata<<=1;                //移位
    MicrowareClock();                //构造时钟
}
}

```

4. 字节接收函数

字节接收函数 **MicrowareReceive** 用于从 Microware 总线上的从器件中接收一个字节的数。程序中使用 for 循环语句来循环接收各位数据，并移位构成一个完整的字节。字节接收函数 **MicrowareReceive** 的程序代码示例如下：

```

uchar MicrowareReceive(void)
{
    uchar receivedata=0;              //接收的字节数据
    int i;
    MicrowareClock();                //构造时钟
    for(i=0;i<8;i++)                 //循环接收
    {
        receivedata*=2;
        if(DO) receivedata++;         //判断 DO，接收位数据
        MicrowareClock();
        Delay(2);
    }
    return receivedata;               //返回数据
}

```

5. 读数据函数

读数据函数 **MicrowareRead** 用于从 Microware 总线上的从器件中读取数据。程序中首先使用字节输入函数发送读指令，接着发送地址，然后通过字节接收函数来接收数据。该函数将返回指定地址中的数据。读数据函数 **MicrowareRead** 的程序代码示例如下：

```

uchar MicrowareRead(uchar addr)
{
    uchar ReadData;
    CS=1;                            //选通
    MicrowareSend(READCode);         //READ 指令
    MicrowareSend(addr);             //指定地址
    ReadData=MicrowareReceive();     //接收数据
    CS=0;                            //选通禁止
    return ReadData;                 //返回数据
}

```

其中，READCode 对应 AT93C66 的 READ 指令码，addr 为指定的地址。

6. 擦写允许函数

擦写允许函数 **MicrowareEnable** 用于将 Microware 总线上的从器件中置于擦除/写允许状态。程序通过向 AT93C66 发送 EWEN 指令码来实现。擦写允许函数 **MicrowareEnable** 的程序代码示例如下：

```

void MicrowareEnable(void)
{
    CS=1;                            //选通
    MicrowareSend(EWENCodeH);        //EWEN 指令
    MicrowareSend(EWENCodeL);
    CS=0;                            //选通禁止
}

```

7. 擦除函数

擦除函数 `MicrowareErase` 用于擦除 `AT93C66` 中指定地址中的数据。程序使用字节输入函数发送 `ERASE` 指令，接着发送地址，然后判断是否擦除完毕。擦除函数 `MicrowareErase` 的程序代码示例如下：

```
void MicrowareErase(uchar addr)
{
    DO=1;
    CS=1;                                //选通
    MicrowareSend(ERASECode);            //ERASE 指令
    MicrowareSend(addr);                 //地址
    CS=0;                                //选通禁止
    Delay(4);
    CS=1;                                //选通
    while(!DO)                           //判断擦除完毕
    {
    }
    CS=0;                                //选通禁止
}
```

其中，`ERASECode` 对应 `AT93C66` 的 `ERASE` 指令码，`addr` 为指定的地址。

8. 写数据函数

写数据函数 `MicrowareWrite` 用于向指定的地址中写数据。程序首先使用 `MicrowareEnable` 函数置 `AT93C66` 为擦除/写允许状态，接着调用 `MicrowareErase` 函数擦除指定地址的数据。然后，通过 `MicrowareSend` 函数分别发送写指令和地址，来实现写数据操作。写数据函数 `MicrowareWrite` 的程序代码示例如下：

```
void MicrowareWrite(uchar addr,uchar Wdata)
{
    MicrowareEnable();                   //擦除/写允许
    MicrowareErase(addr);                 //擦除对应地址的数据
    CS=1;                                //选通
    MicrowareSend(WRITECode);             //写指令
    MicrowareSend(addr);                  //指定地址
    MicrowareSend(Wdata);                 //写数据
    CS=0;                                //选通禁止
    Delay(4);
    CS=1;                                //选通
    Delay(4);
    while(!DO);                           //查询 DO 引脚等待写完
    {
    }
    CS=0;                                //选通禁止
}
```

其中，`WRITECode` 对应 `AT93C66` 的 `WRITE` 指令码，`addr` 为指定的地址。

34.3.4 主程序

这里通过调用前面的 `Microware` 串行总线读写子函数来实现对 `AT93C66` 的操作。其中，在程序中指定 `CS` 接 `P2.0` 引脚，`SK` 接 `P2.3` 引脚，`DI` 接 `P2.2` 引脚，`DO` 接 `P2.4` 引脚。主函数中首先初始化串行口为模式 1，波特率为 4800bit/s。接着通过 `while` 循环语句来扫描串口输入，

根据输入数据来调用函数执行不同的功能。

为了程序设计的方便，这里规定 AT93C66 的指令格式为“00001××A8”。其中，“1”为起始位，“××”为指令的操作码，“A8”为第 9 位地址码。这样，AT93C66 的指令可以表示为如下所示的几种形式。

- 读 (READ): 00001100, 即 READCode = 0x0C。
- 写 (WRITE): 00001010, 即 WRITECode = 0x0A。
- 擦除 (ERASE): 00001110, 即 ERASECode = 0x0E。
- 擦/写允许 (EWEN): 0000100110000000, 即 0x0980。该指令需要分两次写入, 分别是 EWENCodeH=0x09 和 EWENCodeL=0x80。

主函数的程序代码示例如下:

```
#include <reg52.h>                                //头文件
#include <stdio.h>

#define uchar unsigned char
#define High 1                                     //定义高电平
#define Low 0                                     //定义高电平
#define READCode 0x0C                             //AT93C66 读指令
#define WRITECode 0x0A                             //AT93C66 写指令
#define ERASECode 0x0E                             //AT93C66 擦除指令
#define EWENCodeH 0x09                             //AT93C66 擦/写允许指令
#define EWENCodeL 0x80

sbit CS=P2^0;                                     //CS 为 P2.0
sbit SK=P2^3;                                     //SK 为 P2.3
sbit DI=P2^2;                                     //DI 为 P2.2
sbit DO=P2^4;                                     //DO 为 P2.4

void Delay(uchar n);                             //延时
void MicrowareClock(void);                       //时钟函数
void MicrowareSend(uchar senddata);             //字节输入函数
uchar MicrowareReceive(void);                   //字节接收函数
uchar MicrowareRead(uchar addr);                //读数据函数
void MicrowareEnable(void);                     //擦写允许函数
void MicrowareErase(uchar addr);                //擦除函数
void MicrowareWrite(uchar addr,uchar Wdata);    //写数据函数

void main()
{
    uchar Select_Menu;                           //功能选择
    uchar rd;
    SCON=0x50;                                   //初始化串行口模式 1
    TMOD=0x20;                                   //初始化 T1 为定时功能, 模式 2
    PCON=0x80;                                   //设置 SMOD=1
    TL1=0xF4;                                    //波特率 4800bit/s, 初值
    TH1=0xF4;

    TR1 = 1;                                     //启动 T1
    TI = 1;                                     //启动发送
```

51 单片机开发与应用技术详解

```
while(1)
{
    printf("Microware AT93C66 Write and Read!");
    printf (" 1. Enable AT93C66.\n"); //菜单信息
    printf (" 2. Read AT93C66.\n");
    printf (" 3. Write AT93C66.\n");
    printf (" 4. Erase AT93C66.\n");
    printf (" 5. EXIT.\n");
    printf (" Please Input Menu Option:\n");

    Select_Menu = _getkey(); //从键盘输入选择数字
    switch(Select_Menu)
    {
        case '1': //擦写使能 AT93C66
            printf ("\n You Select 1. Enable AT93C66\n");
            MicrowareEnable();
            printf ("\n Enable over\n");
            break;

        case '2': //读 AT93C66
            printf ("\n You Select 2. Read AT93C66\n");
            rd=MicrowareRead(0x12);
            printf("AT93C66 data at 0x12=%d\n",rd);
            break;

        case '3': //写 AT93C66
            printf("\n You Select 3. Write AT93C66\n");
            MicrowareWrite(0x12,0x56);
            printf ("\n Write over\n");
            break;

        case '4': //搜索擦除 AT93C66
            printf ("\n You Select 4. Erase AT93C66\n");
            MicrowareErase(0x12);
            printf ("\n Erase over\n");
            break;

        case '5': //退出程序
            printf ("\n You Select 5. Exit\n");
            goto Exit; //转向 Exit 标号处
            break;

        default:
            printf ("\n Error: Please Select Right Menu Option\n");
            break;
    };
}

Exit: printf("Exit the program!"); //退出
}
```

通过前面的子函数及这里的主程序,便可以实现 Microware 总线接口的 AT93C66 的读写操作。

34.3.5 Microware 串行总线仿真

Keil μ Vision3 集成开发环境提供了很好的信号仿真功能,下面就利用其进行程序的仿真分析。具体操作步骤如下:

- (1) 在 Keil μ Vision3 集成开发环境中,选择“Debug”→“Start/Stop Debug Session”命令,进入仿真分析模式。
- (2) 选择“View”→“Serial Window”→“UART #0”命令,打开串口仿真接口。
- (3) 选择“View”→“Logic Analyzer Window”命令,打开波形仿真界面。
- (4) 单击“Setup”按钮,打开“Setup Logic Analyzer”对话框,在其中输入待仿真的信号 CS、SK、DI 和 DO,如图 34-13 所示。

(5) 选择 “Debug” → “Go” 命令，开始执行仿真。

(6) 此时，在串行仿真接口中便输出读写 AT93C66 的控制菜单。这里输入 “3”，即选择写 AT93C66。程序便执行写操作，如图 34-14 所示。

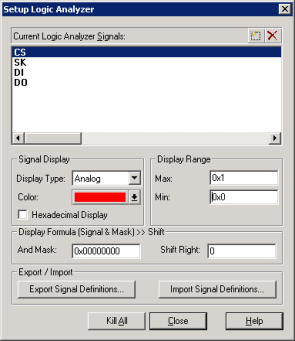


图 34-13 输入仿真信号

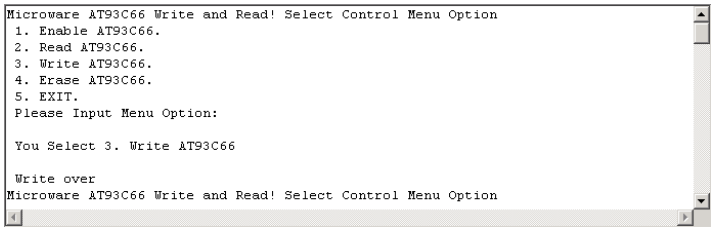


图 34-14 仿真结果

此时，在波形仿真窗口便可以看到指定的 Microware 信号的仿真波形，如图 34-15 所示。读者可以通过其波形操作和前面介绍的指令波形对比来判断程序执行的正确性。

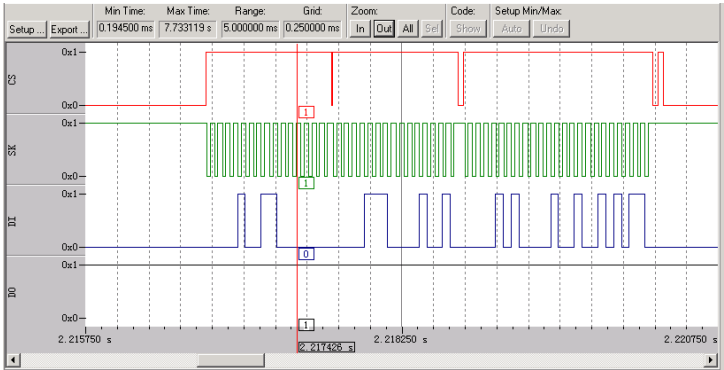


图 34-15 仿真波形

当仿真无误后，便可以将程序下载到单片机中执行。

34.4 小结

本章详细讲解了三线制 Microware 串行总线，并通过 Microware 串行总线的 EEPROM 介绍了 Microware 串行总线的操作指令及其操作时序。本章最后通过一个具体的实例，介绍了如何使用标准的 51 系列单片机来在软件上仿真模拟 Microware 串行总线。其中给出了 Microware 串行总线读写的子函数及一个完整的实例。三线制 Microware 串行总线减少了 I/O 引脚的使用，在实际电路设计中具有广泛的应用，读者应该熟练掌握。

第 35 章 单片机控制打印机实例

在单片机应用系统中，经常需要用到信息的打印输出。单片机可以通过外接的打印机来输出运行状态、测量结果及格式化输出数据等。目前，市场上打印机有很多种。按照打印原理可以分为键式打印机、针式打印机、热敏打印机、喷墨式打印机、激光打印机等。按照打印的行宽可以分为宽行打印机、窄行打印机、微型打印机。按打印头是否能往返打印可以分为单向打印机和双向打印机。按打印的字符颜色可以分为单色打印机和彩色打印机。

微型打印机体积小巧、接口简单、控制灵活。因此，微型打印机最适合与单片机相连接。本章将主要介绍如何使用 51 系列单片机来控制微型打印机进行打印输出。

35.1 打印机概述

目前，常用的微型打印机均采用规范化的 Centronics 并行接口标准。Centronics 标准可以方便地和计算机及微处理器的并行端口连接。目前，市场上应用最为广泛的微型打印机有如下几种。

- LASER PP40 描绘器；
- TP-UP-16A 智能点阵式打印机；
- TP-UP-40A 智能点阵式打印机；
- GP16 微型打印机；
- XLF 嵌入式汉字微型打印机。

这些微型打印机均提供了丰富的指令，具有多种打印模式，可用支持字符及图形的打印。微型打印机作为智能外设，可用来方便地与各种并行端口连接，可用来永久地保存数据，因此应用十分广泛。下面以 LASER PP40 为例，介绍如何使用 51 系列单片机来实现打印机控制。

35.1.1 LASER PP40 打印机概述

LASER PP40 是一种四色描绘式微型打印机，其具有文本模式和图形模式两种工作方式。可用来描绘字符及图形，具有较强的绘图功能。LASER PP40 与微处理器相结合，便可以在智能仪表及实时控制系统中作为微型绘图机使用。LASER PP40 的主要特点如下：

- 标准的 Centronics 并行接口；
- 丰富的打印指令，格式简单；
- 支持全部的 ASCII 码字符，以及常用的控制字符；
- 具有较强的绘制图形能力。

LASER PP40 和外部的接口包括 36 根 I/O 引脚，其引脚定义如表 35-1 所示。

表 35-1 PP40 接口信号

引 脚	信 号	引 脚	信 号	引 脚	信 号	引 脚	信 号
1	STROBE	10	ACK	19	GND	28	GND
2	DATA1	11	BUSY	20	GND	29	GND

3	DATA2	12	GND	21	GND	30	NC
4	DATA3	13	NC	22	GND	31	NC
5	DATA4	14	GND	23	GND	32	GND
6	DATA5	15	GND	24	GND	33	NC
7	DATA6	16	GND	25	GND	34	NC
8	DATA7	17	GND	26	GND	35	NC
9	DATA8	18	NC	27	GND	36	NC

LASER PP40 微型打印机的各个引脚信号满足 Centronics 标准,其中所有 I/O 信号均与 TTL 电平兼容。LASER PP40 各引脚含义如下:

- STROBE (Pin1): 打印机选通输入信号。LASER PP40 在 STROBE 的上升沿丢入 DATA1~DATA8 上的信息,并启动其内部的机械装置开始打印操作。
- DATA1~DATA8 (Pin2~Pin9): 数据信号。
- ACK (Pin10): 握手响应信号。当 LASER PP40 完成一次数据处理及打印操作后,ACK 引脚便输出一个负脉冲。
- BUSY (Pin11): 忙状态输出信号。如果 LASER PP40 正在处理打印命令或数据,BUSY 引脚将一直输出高电平,当 LASER PP40 空闲时将一直输出低电平。
- GND: 接地信号。其中部分引脚需要和信号线相绞,以提高抗干扰能力,增大连线长度。
- NC: 空引脚。

在 Centronics 标准定义的信号线中,最主要的是 8 位并行数据线、选通输入信号、握手响应信号线和忙状态信号线 BUSY。LASER PP40 工作时序,如图 35-1 所示。

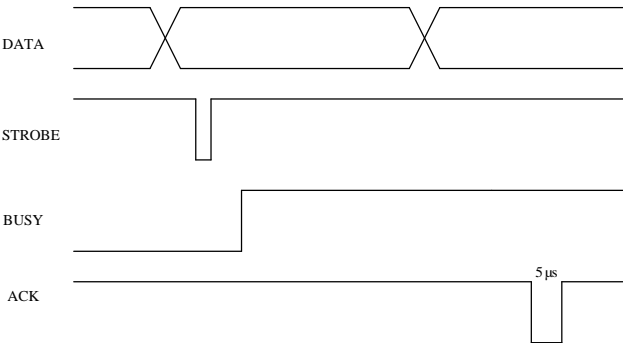


图 35-1 LASER PP40 的工作时序

当 CPU 通过接口要求打印机打印数据时,先要查看忙信号 BUSY。如果 BUSY=0,即不忙才能向打印机输出数据,否则不能向打印机输出数据。在把数据送到 DATA 线上后,先发出选通信号通知打印机;打印机收到选通信号后,先发出“忙”信号,再从接口接收数据。当数据接收完并存入内部的打印缓冲器后,便送出 ACK 响应信号(宽度为 5 μs 的负脉冲),表示打印机已准备好接收数据。同时在 ACK 脉冲的后沿使 BUSY 信号撤销。

35.1.2 LASER PP40 的文本模式

LASER PP40 具有文本模式和图形模式两种操作方式,上电后初始状态为文本模式。LASER PP40 在文本模式下可打印输出的 ASCII 字符,如表 35-2 所示。

表 35-2 PP40 可打印的 ASCII 字符

	0	1	2	3	4	5	6	7
0				0	@	P	`	p
1	☉	♂	0	1	A	Q	a	q
2	●	♀	”	2	B	R	b	r
3	♥		#	3	C	S	c	s
4	♦		\$	4	D	T	d	t

5	♣		%	5	E	U	e	u
6	♠		&	6	F	V	f	v
7	●		'	7	G	W	g	w
8	■		(8	H	X	h	x
9)	9	I	Y	i	y
A	LF		*	:	J	Z	j	z
B	LU		+	;	K	[k	}
C			,	<	L	\	l	
D	CR	NC	-	=	M]	m	{
E	♪		.	>	N	^	n	~
F	☼		/	?	O	-	o	×

表中除了列出一些字符编码，还列出了一些控制编码，其含义如下：

- 回位（08H）：使描绘笔回到前面一个字符位置，若描绘笔已处于该行最左边位置，则该命令无效。
 - 进纸（0AH）：将打印纸推进一行。
 - 退纸（0BH）：将打印纸退后一行。
 - 回车（0DH）：使描绘笔回到行的最左边，并进纸一行。
 - 控制 1（11H）：与回车符配合使用，可以将 LASER PP40 置为文本模式。
 - 控制 2（12H）：与回车符配合使用，可以将 LASER PP40 置为图案模式。
 - 转色（1DH）：使描绘笔转动一个位置，更换另外一种颜色的描绘笔。
- 当打印输出超过一行的字数后，LASER PP40 自动回车并进纸一行。

35.1.3 LASER PP40 的图形模式

当 LASER PP40 处于文本模式时，外部控制器将回车符（0DH）和控制 1（11H）写入 LASER PP40，则由文本模式转换成图形模式；再将回车符（0DH）和控制 2（12H）写入 LASER PP40，则又回到文本模式。

LASER PP40 在图形模式工作时，可选择各种绘图操作命令，以便绘出各种图形、表格、曲线。绘图命令共 13 条，分为如下所示的几类。

不带参数的单字符命令，这类命令包括 A、H 和 I 命令，分别介绍如下：

- 重置：命令格式为“A”。LASER PP40 笔架返回 X 轴最左方，而 Y 轴不变动。返回文字模式并以笔架停留处作为起点。
- 回档：命令格式为“H”。LASER PP40 笔嘴升起返回起点。
- 预备：命令格式为“I”。LASER PP40 以笔架位置作为起点。

只带一个参数的命令，这类命令包含 L、C、S、Q4 条命令，参数跟在命令符号后面，分别介绍如下：

- 线形式：命令格式为“LP（由 0 至 15）”。LASER PP40 所绘画线的形式。其中，实线为 P=0，点线 P=由 1~15，分别具有不同的格式含义。
- 颜色转换：命令格式为“Cn（n=由 0 至 3）”。LASER PP40 所使用的颜色。颜色转换由 n 所指定，其中 0 表示黑色，1 表示蓝色，2 表示绿色，3 表示红色。
- 字符尺码：命令格式为“Sn（n=由 0 至 63）”。指定 LASER PP40 字符尺码。
- 字母绘制方向：命令格式为“Qn（n=由 0 至 3）”。指定 LASER PP40 文字编印方向（只在图形模式下适用）

带两个参数的命令，这类命令包括 D、J、M、R4 条命令。指令参数之间需以“，”作分隔，指令以回车（0DH）结束，分别介绍如下：

- 绘线：命令格式为“Dx,y……X,Yn（-999≤x,y≤999）”。LASER PP40 由当前笔嘴位置到坐标点（X,Y）连线。
- 相对绘线：命令格式为“J△x, △y……△Xn,Yn（-999≤△x, △y≤999）”。LASER PP40

由当前笔嘴位置连线至笔嘴点 Δx , Δy 距离之点上。

- 移动：命令格式为“ Mx,y ($-999 \leq x,y \leq 999$)”。LASER PP40 笔嘴升起，移动至起点相距 Δx , Δy 之新点上。
- 相对移动：命令格式为“ $R\Delta x, \Delta y$ ($-999 \leq \Delta x, \Delta y \leq 999$)”。LASER PP40 笔嘴升起，移动至当前笔架相距 Δx , Δy 之新点上。

P 指令，命令格式为“PC, C……Cn (n 无限制)”。用以编绘字符，字符与字符间以“,” 分隔，以回车结束。

X 指令，命令格式为“ Xp,q,r (p=由 0 至 1) (q=-999 至 999) (r=1 至 255)”。用以绘制坐标及分度线，带有 3 个参数。参数之间以“,” 分隔，以回车结束。由当前笔架位置绘画轴线，其中，Y 轴为 p=0, X 轴为 p=1, q 为点距, r 为重复次数。

35.2 51 系列单片机控制打印机实例

LASER PP40 接口满足 Centronics 标准，可以通过单片机来实现打印输出控制。下面介绍如何使用 51 系列单片机来控制 LASER PP40 微型打印机。

35.2.1 电路图

这里采用 AT89S52 单片机来控制 LASER PP40 微型打印机。系统电路原理图，如图 35-2 所示。该电路中所使用的元器件清单，如表 35-3 所示。

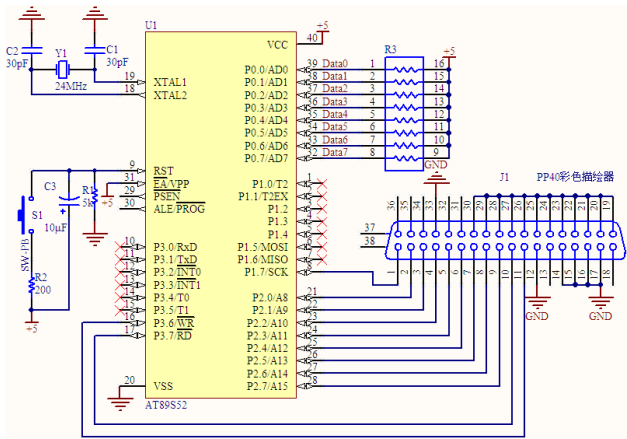


图 35-2 电路原理图

表 35-3 元器件列表

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89S52	1 个	R1	5k Ω	1 个
J1	LASER PP40	1 个	R2	200 Ω	1 个
C1、C2	30pF	2 个	R3	10k Ω 排阻	1 个
C3	10 μ F 电解	1 个	Y1	24MHz	1 个
S1	按键开关	1 个			

其中，使用了单片机 AT89S52 的 P2 端口连接 LASER PP40 微型打印机的 DATA, P1.7 引脚连接 STROBE, P3.6 引脚连接 BUSY, P3.7 引脚连接 ACK。

35.2.2 建立项目

- 首先在 Keil μ Vision3 集成开发环境中建立项目，具体操作步骤如下：
- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project” → “New” → “ μ Vision Project”

51 单片机开发与应用技术详解

命令，新建一个工程，并保存。

(2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 35-3 所示。

(3) 单击“确定”按钮，此时弹出“μ Vision3”对话框，如图 35-4 所示。单击“是”按钮，完成工程的建立。

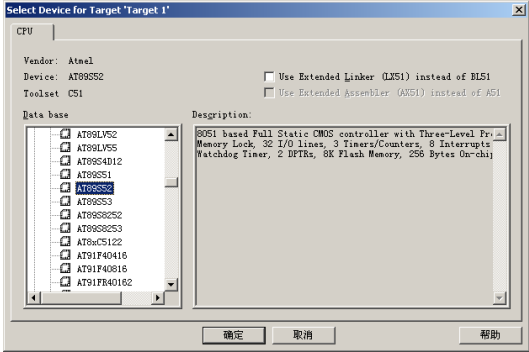


图 35-3 选择单片机 AT89S52



图 35-4 “μ Vision3”对话框

(4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

35.2.3 程序设计

单片机控制 LASER PP40 微型打印机打印输出的程序代码，示例如下：

```
#include<reg52.h>                                //头文件

sbit STROBE=P1^7;                                //STROBE 引脚
sbit BUSY=P3^6;                                  //BUSY 引脚
sbit ACK=P3^7;                                   //ACK 引脚

unsigned char code Message[]=                    //一行消息
    {0x48,0x65,0x6c,0x6c,0x6f,0x20,0x77,0x6f,0x6c,0x6d,0x21};

void PP40Char(unsigned char ch)                  //PP40 打印单个字符
{
    P1=ch;                                        //字符输出到 DATA
    STROBE=0;                                    //STROBE 低电平
    STROBE=1;                                    //STROBE 上升沿
    while(BUSY);                                //查询等待 PP40 打印结束
}

void PP40Line(void)                              //PP40 打印一行
{
    unsigned char i;
    for (i=0;i<=10;i++)                          //循环打印输出一行信息
    {
        P1=Message[i];                          //字符输出到 DATA
        STROBE=0;                                //STROBE 低电平
        STROBE=1;                                //STROBE 高电平
        while(BUSY);                             //查询等待 PP40 打印结束
    }
}
```

```

void main(void)
{
    int i;
    for(i=0;i<10;i++)
    {
        PP40Char(0x2A);           //打印输出“*”
    }
    PP40Char(0x0D);               //换行
    PP40Line();                   //打印输出一行消息“Hello world!”
    PP40Char(0x0D);               //换行
    PP40Char(0x57);               //打印输出字符“W”
    PP40Char(0x45);               //打印输出字符“E”
    PP40Char(0x4C);               //打印输出字符“L”
    PP40Char(0x43);               //打印输出字符“C”
    PP40Char(0x4F);               //打印输出字符“O”
    PP40Char(0x4D);               //打印输出字符“M”
    PP40Char(0x45);               //打印输出字符“E”
    PP40Char(0x0D);               //换行
    while(1)                       //空循环
    {
    }
}

```

其中，定义了两个自定义函数 **PP40Char** 和 **PP40Line**。**PP40Char** 用于打印输出一个单个字符，**PP40Line** 用于打印输出一行消息。在主函数中，分别使用这两个自定义函数来控制 **LASER PP40** 打印输出。最终将在纸上打印输出如下信息。

```

*****
Hello world!
WELCOME

```

35.3 小结

本章详细讲述了 **LASER PP40** 微型四色描绘式打印机，包括其接口、工作时序、文本模式和图形模式等。本章最后还通过一个具体的实例，介绍了如何使用 **51** 系列单片机控制 **LASER PP40** 微型打印机打印输出。**LASER PP40** 微型打印机接口简单、控制方便，十分适合作为单片机系统的智能输出扩展。因此，熟练掌握本章内容对读者以后的单片机系统设计工作很有帮助。

第 36 章 A/D 转换实例

在测控系统中，经常需要对温度、速度、压力、电流、电压等模拟量进行采集或处理。由于单片机 CPU 只能对数字信号进行处理，因此，需要首先将这些模拟量信号转换成数字量信号，然后采集数据并进行分析。这便需要用到模/数转换器件，也称为 A/D（Analogue/Digital）转换器。目前，市场上有很多种 A/D 转换器，其以体积小、功能强、误差小、功耗低、可靠性高等优点而得到广泛应用。

本章将主要介绍 A/D 转换的原理，A/D 转换器的技术参数，并将介绍一个典型的 8 通道电压型 A/D 转换器。最后，本章将通过一个具体的实例，介绍如何使用 51 系列单片机来控制 A/D 转换器的读写。

36.1 A/D 转换概述

数据采集分为数字量采集和模拟量采集。其中，数字量采集只需采集二进制的数字信号即可；而模拟量采集则需要首先将待采集的模拟信号转换成数字信号，然后再进行采集及后续分析处理。在模拟量采集的过程中，A/D 转换是一个重要的环节。一个完整的模拟量采集系统应该包括信号调制电路、采样/保持放大器、模拟/数字（A/D）转换器、通道控制电路和采集电路等。下面主要介绍 A/D 转换器的原理及应用。

36.1.1 A/D 转换原理

模拟/数字（A/D）转换器的功能是将输入的模拟信号转换成数字信号的形式输出。其中，输入的模拟信号一般为电压或电流。对于需要采集其他模拟量的场合，例如速度、气压和温度等，需要首先使用相应的传感器将这些模拟信号转换成电压或电流的模拟信号，然后再选择合适的 A/D 转换器。

目前市场上 A/D 转换器的种类很多，按照 A/D 转换的原理主要有两类，即逐次渐近型 A/D 转换器和双积分型 A/D 转换器。下面以电压型 A/D 转换器为例，来介绍这两种 A/D 转换器的基本原理。

1. 逐次渐近型 A/D 转换器原理

逐次渐近型 A/D 转换器主要由 D/A 转换器、电压比较器、锁存器、移位寄存器和控制逻辑等部分组成。8 位逐次渐近型 A/D 转换器原理图，如图 36-1 所示。

逐次渐近型 A/D 转换器的工作原理是，待测的模拟电压 U_{Input} 输入电压比较器的 A 端口，D/A 转换器输出的电压 $U_{Digital}$ 输入电压比较器的 B 端口。A 端口和 B 端口的数据进行比较，根据 B 端口的电压是大于还是小于 A 端口的电压，来输出反馈信号，以便使 D/A 转换器输出的模拟电压逐次逼近实际的模拟电压 U_{Input} 。当 D/A 转换器输出的电压 $U_{Digital}$ 和模拟电压 U_{Input} 相等的时候，A/D 转换结束，此时 D/A 转换器输入的数字量便是对应模拟电压的数字量。

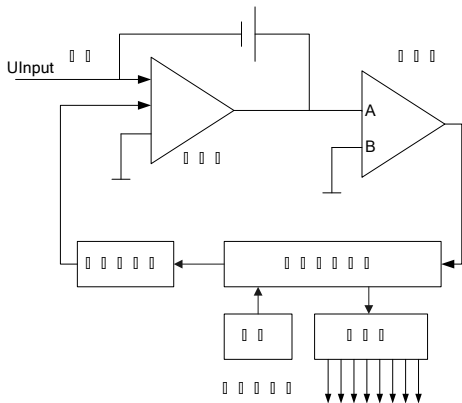


图 36-2 双积分式 A/D 转换器原理图

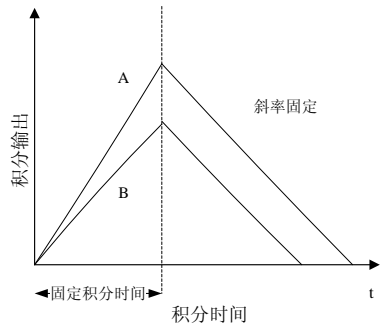


图 36-3 积分示意图

36.1.2 A/D 转换器的技术参数

不同的 A/D 转换器具有不同的性能，在选择使用 A/D 转换器时，需要了解 A/D 转换器的相关性能指标及技术参数。下面列出几个关键的技术参数，供读者参考。

1. 分辨率

分辨率是指 A/D 转换器的最小分辨能力。分辨率通常采用输出数字量的二进制位数来表示，如 8 位、10 位、12 位、14 位和 16 位等。如果 A/D 转换器的分辨率为 N，则表示其可将整个量程分为 2^N 份，最小可以分辨 $1/2^N$ 的增量。例如，对于 12 位的 A/D 转换器，最小的分辨能力如下：

$$1\text{LSB} = 1/2^{12} \times 100\% = 1/4096 \times 100\% = 0.0244\%$$

分辨率越高，A/D 转换过程中对输入量的微小变化的反应越灵敏。

2. 量程

量程是 A/D 转换器所测量的模拟量的范围。对于电压型 A/D 转换器，典型的量程范围如 0~5V、0~10V、-5V~5V、-10V~10V 等。

3. 转换精度

转换精度是 A/D 转换器转换后所得结果相对于实际值的准确度。A/D 转换器的转换精度有绝对精度和相对精度两种，反映了一个实际的 A/D 转换器与理想 A/D 转换器进行 A/D 转换的差值。常用数字量的位数作为度量精度的单位，如精度为 $\pm 1/2\text{LSB}$ ，用百分比来表示满量程时的相对误差，如 $\pm 0.05\%$ 。

这里需要指出的是，精度和分辨率是两个不同的概念。精度指的是转换结果与实际值的准确程度，而分辨率是指相对转换结果发生影响的最小输入量。分辨率可以很高，但可能由于系统设计的原因导致温度漂移等而并不具有很高的精度。

4. 转换时间

转换时间是指 A/D 转换器完成一次 A/D 转换所需要的时间。转换时间的倒数即转换速率。例如，高速电压型 A/D 转换器转换时间为 20ns~50ns，即转换速率可达 20~50MSPS。

5. 温度系数

温度系数是 A/D 转换器的温度表现能力。A/D 转换器很容易受环境温度影响，其温度系数主要有失调（零点）温度系数和增益温度系数。温度系统一般用每摄氏度温度变化所产生的相对误差来衡量，以 ppm/℃ 为单位。一般 A/D 转换器均标有工作温度范围，也就是说在该温度范围内，可以确保给出的 A/D 转换性能指标。

6. 对电源电压变化的抑制比

对电源电压变化的抑制比（PSRR）是指 A/D 转换器对电源电压的依赖性。其一般用改变电源电压使数据发生 $\pm 1\text{LSB}$ 变化时，所对应的电源电压变化范围来表示。

36.1.3 A/D 转换器的选择原则

采用 A/D 转换器的模拟信号采集是一个要求比较高的工作，需要考虑多方面的问题。下面介绍需要重点注意的几个问题。

- 采样速度。采样速度决定了数据采集系统的实时性。采样速度由模拟信号带宽、数据通道数和每个周期的采样数来决定。采集速度越高，对模拟信号复原的越好，也即实时性越好。根据奈奎斯特采样定理可知，数据采集系统对源信号无损再现的必要条件是，采样频率至少为被采样信号最高频率的两倍。
- A/D 转换精度。对于复杂系统，一般计算系统中各环节的均方根误差。信号源阻抗、信号带宽、A/D 转换器分辨率和系统的通过率都会影响误差的计算。正常情况下，A/D 转换前向通道的总误差应小于等于 A/D 转换器的量化误差，否则选取高分辨率 A/D 转换器也没有实际意义。
- 孔径误差。A/D 转换是一个动态的过程，需要一定的转换时间。而输入的模拟量总是在连续不断变化的，这样便造成转换输出的不确定性误差，即孔径误差。为了确保较小的孔径误差，则要求 A/D 转换器具有与之相适应的转换速度。否则，就应该在 A/D 转换器前加入采样/保持电路以满足系统要求。
- 系统通过率。系统的通过率由模拟多路选择器、输入放大器的稳定时间、采样/保持电路的采集时间及 A/D 转换器的转换时间确定。在模拟信号的数据采集系统中，有正常顺序和重叠两种采集方式，采用低分辨率 A/D 转换器、减少模/数转换环节及采用重叠方式采集时，可获得较大带宽的通过速度。

在设计模拟信号的采集系统时，A/D 转换器的选型是一个重要环节。一般来说，在选择 A/D 转换器时，需要遵循如下原则。

- 根据输入模拟信号的变化速率，确定 A/D 转换器的转换速度，从而保证整个模拟信号数据采集系统的实时性。
- 选择 A/D 转换器的精度和分辨率时，需要考虑前向系统的总误差，将综合精度在各个环节上进行再分配，以确定 A/D 转换器的精度要求。
- 根据需要，对变化比较快的模拟信号可以考虑使用采用/保持电路，这样可以在进行 A/D 转换时，减少孔径误差。
- 根据工作现场环境条件选择 A/D 转换器的一些环境参数要求，如工作温度范围、湿度范围、可靠性等级、电源电压抑制比、功耗等。
- 根据采集模拟信号的数量，来选择多个通道的 A/D 转换器。
- 根据 A/D 转换器和单片机或其他微处理器的接口，来合理选择 A/D 转换器的输出状态，例如 TTL 或 CMOS 电平、串行输出还是并行输出等。
- 一般来说，每一款 A/D 转换器均提供了多种封装供选择。使用时需要根据电路板的面积等，选择合适封装的 A/D 转换器。

36.2 8 通道 A/D 转换器 MAX197

目前，各大半导体公司均推出了完善的 A/D 转换产品系统，其使用方法大致类似。下面介绍一款性能比较优越的多通道 A/D 转换器 MAX197。

36.2.1 MAX197 的特性及引脚功能

MAX197 是美国 MAXIM 公司生产的 8 通道 12 位电压型 A/D 转换器。MAX197 具有 5MHz

51 单片机开发与应用技术详解

的带宽，100kSPS 的高速数据吞吐量。MAX197 的芯片工作电压仅为+5V，能够和典型的单片机系统很好地兼容。MAX197 提供了多种测量量程，可以接收高于电源电压的模拟电压信号，也可以接收低于系统地电压的模拟电压信号，使用灵活方便。MAX197 的主要特性如下：

- 单一+5V 供电；
- 12 位 A/D 转换分辨率，1/2LSB 线性度；
- 12 位并行输出，引脚兼容 TTL/CMOS 电平；
- 8 路模拟输入通道；
- 4 种软件可编程输入量程：0~+5V、0~+10V、-5V~+5V、-10V~+10V；
- 6μs 典型转换时间，高达 100kSPS 的采样速率；
- 内部集成 4.096V 参考电压，也可以采用外部参考电压；
- 可选择内部工作时钟或外部工作时钟；
- 可选择使用内部采集控制或外部采集控制；
- 软件可编程两种掉电工作模式。

MAX197 的引脚图，如图 36-4 所示。下面介绍 MAX197 各个引脚的功能。

- CLK (Pin1)：时钟输入引脚。在采用外部时钟模式时，由此引脚输入时钟信号。在采用内部时钟模式时，该引脚和地之间接一个电容，以确定内部时钟频率。

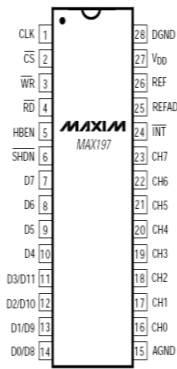


图 36-4 MAX197 引脚分配图

- CS (Pin2)：片选信号输入引脚，低电平有效。
- WR (Pin3)：在内部采集模式下，WR 的上升沿将锁存数据，并启动一次采集和一次转换周期；在外部采集模式下，WR 的第一个上升沿启动采集，第二个上升沿结束采集并启动转换周期。
- RD (Pin3)：RD 的下降沿将允许读取数据总线上的数据。
- HBEN (Pin5)：用于切换 12 位的数据转换输出结果。当 HBEN=1 时，数据总线上为高 4 位；当 HBEN=0 时，数据总线上为低 8 位。
- SHDN (Pin6)：关断控制位。当 SHDN=0 时，器件进入掉电模式 (FULLPD)。MAX197 的 SHDN 脚和两个软件可编程位 (STBYPD、FULLPD) 用来提供转换工程中的低电流关断模式。

- D7~D4 (Pin7~Pin10)：三态 I/O 接口。
- D3/D11 (Pin11)：三态 I/O 接口。当 HBEN=0 时，输出 D3；当 HBEN=1 时，输出 D11。
- D2/D10 (Pin12)：三态 I/O 接口。当 HBEN=0 时，输出 D2；当 HBEN=1 时，输出 D10。
- D1/D9 (Pin13)：三态 I/O 接口。当 HBEN=0 时，输出 D1；当 HBEN=1 时，输出 D9。
- D0/D8 (Pin14)：三态 I/O 接口。当 HBEN=0 时，输出 D0；当 HBEN=1 时，输出 D8。
- AGND (Pin15)：模拟地。
- CH0~CH7 (Pin16~Pin23)：8 个模拟电压输入通道。
- INT (Pin24)：当转换完毕，且数据准备就绪的时候，INT 引脚电平变低。
- REFADJ (Pin25)：内部参考电压输出和外部调节引脚。使用内部参考电压时，该引脚对地接 0.01μF 的旁路电容；使用外部参考电压时，该引脚接 VDD。
- REF (Pin26)：内部参考电压缓冲输出或 ADC 参考电压输入。在内部参考电压模式下，该引脚输出一个 4.096V 的标准电压，可由 REFADJ 引脚进行外部调节；在外部参考电压模式下，REFADJ 接到 VDD，内部缓冲器处于禁止状态。
- VDD (Pin27)：外接+5V 电源。
- DGND (Pin28)：数字地。

36.2.2 MAX197 的接口、控制字及时序

A/D 转换器 MAX197 与单片机或其他微处理器接口十分方便，其典型的接线图，如图 36-5 所示。MAX197 工作方式的控制，主要依靠微处理器向 MAX197 发送控制字节来实现。MAX197 控制字节及各位的定义，如表 36-1 所示。

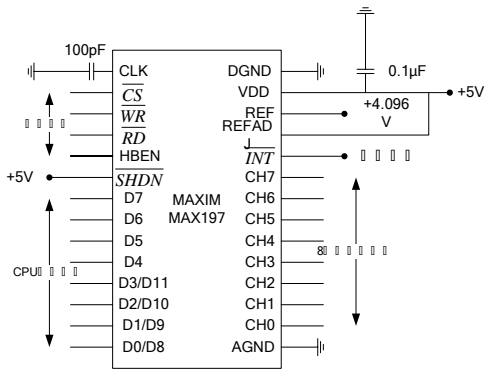


图 36-5 MAX197 与 CPU 的接口电路

表 36-1 MAX197 的控制字节

D7	D6	D5	D4	D3	D2	D1	D0
PD1	PD0	ACQMOD	RNG	BIP	A2	A1	A0

其中，控制字节中的 D7、D6 位控制芯片的时钟模式和低功耗模式，如表 36-2 所示。下面分别介绍 MAX197 的各个工作模式。

表 36-2 时钟、掉电模式选择

PD1	PD0	器件工作模式	PD1	PD0	器件工作模式
0	0	工作状态/外部时钟模式	1	0	待机模式，时钟模式不变
0	1	工作状态/内部时钟模式	1	1	掉电模式，时钟模式不变

➤ 当 D7=0, D6=0 时，MAX197 使用外部时钟模式，外部时钟频率在 100kHz~2.0MHz 之间，时钟周期应介于 45%~55% 之间。外部时钟的时序图，如图 36-6 和图 36-7 所示。

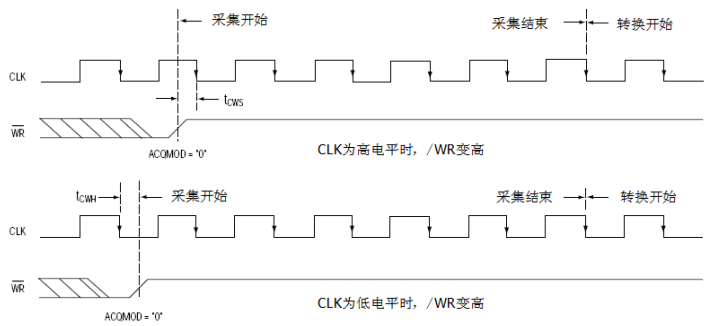


图 36-6 在内部采集控制模式下使用外部时钟和 WR 的时序

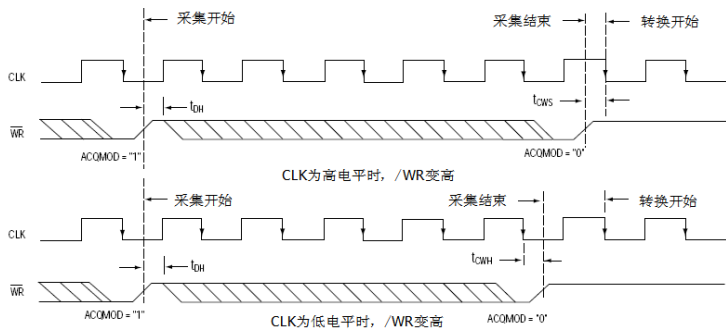


图 36-7 在外部采集控制模式下使用外部时钟和 WR 的时序

- 当 D7=0, D6=1 时, MAX197 选择内部时钟模式, 此时在 CLK 与地之间应接一个 100pF 的电容, 与之相应的采样频率为 $f_{CLK}=1.56\text{MHz}$ 。
 - 当 D7=1, D6=0 时, MAX197 进入待机模式。
 - 当 D7=1, D6=1 时, MAX197 进入掉电模式。
- 控制字节中的 D5 位 ACQMOD 决定 MAX197 控制采集的模式, 分为内部采集控制模式和外部采集控制模式, 下面分别进行介绍。
- 当 ACQMOD=0 时, MAX197 为内部采集控制模式, 时序图如图 36-8 所示。

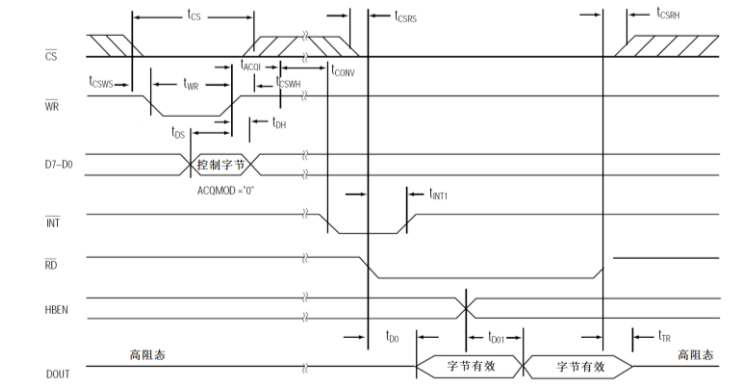


图 36-8 内部采集控制模式时序图

- 当 ACQMOD=1 时, MAX197 为外部采集控制模式, 时序图如图 36-9 所示。

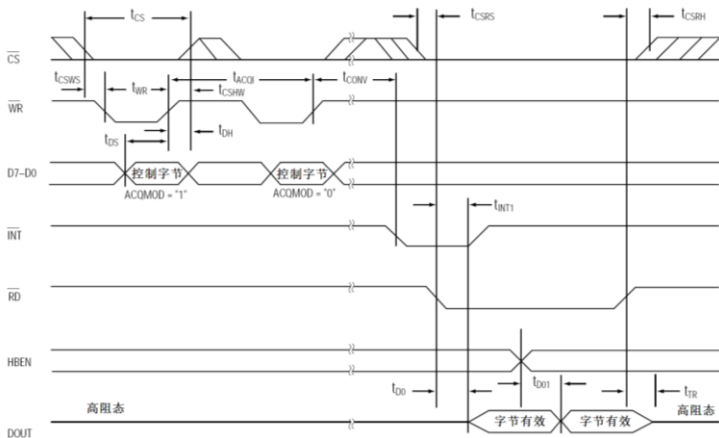


图 36-9 外部采集控制模式时序图

MAX197 控制字节的 D3 和 D4 用来选择芯片的模拟输入电压范围和极性, 如表 36-3 所示。

D3=0，选择单极性输入；D3=1 时，选择双极性输入；D4=0 时，选择 5V 量程；D4=1 时，选择 10V 量程。

表 36-3 量程、极性选择

BIP	RNG	输入量程/V	BIP	RNG	输入量程/V
0	0	0~5	1	0	-5~+5
0	1	0~10	1	1	-10~+10

MAX197 控制字节的低三位 A0~A2 用来选通模拟输入通道。三位数值与被选通模拟输入通道之间的关系，如表 36-4 所示。

表 36-4 模拟输入通道

A2	A1	A0	CH0	CH1	CH2	CH3	CH4	CH5	CH6	CH7
0	0	0	√							
0	0	1		√						
0	1	0			√					
0	1	1				√				
1	0	0					√			
1	0	1						√		
1	1	0							√	
1	1	1								√

36.3 单片机读写 A/D 转换器实例

在实际的测控领域，有很多场合需要使用 A/D 转换和数据采集。下面介绍如何使用 51 系列单片机，结合 8 通道 A/D 转换器 MAX197 来实现模拟电压的采集控制。

36.3.1 电路图

这里采用 Atmel 公司的 AT89C51 单片机来执行控制。整个系统的电路原理图，如图 36-10 所示。电路中所使用的元器件列表，如表 36-5 所示。

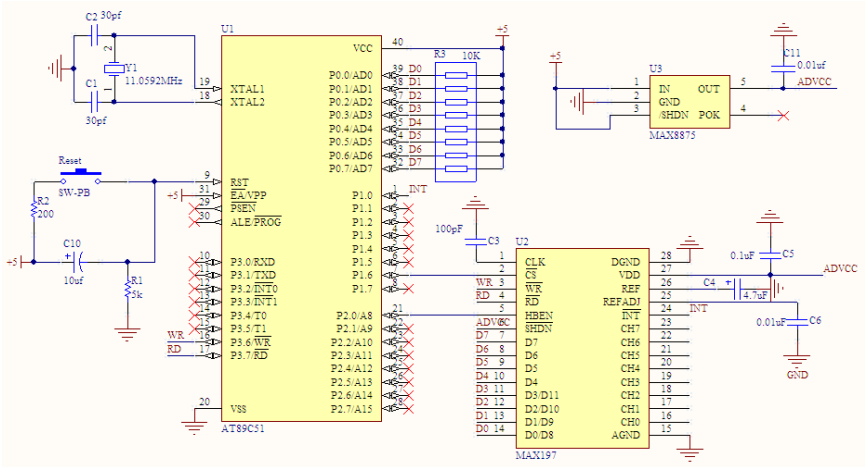


图 36-10 电路图

表 36-5 元器件

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89C51	1 个	C6、C11	0.01 μF	2 个
U2	MAX197	1 个	C10	10 μF 电解电容	1 个

U3	MAX8875	1 个	R1	5k Ω	1 个
C1、C2	30pf	2 个	R2	200 Ω	1 个
C3	100pF	1 个	R3	10k Ω 排阻	1 个
C4	4.7 μ F	1 个	Reset	按键开关	1 个
C5	0.1 μ F 电解电容	1 个	Y1	11.0592MHz	1 个

其中采用 AT89C51 作为控制 CPU,使用 MAX197 作为 A/D 转换采集 8 个通道的模拟电压。由于 MAX197 为 12 位的 A/D 转换器,所以其转换精度可以达到 1/4096,精度非常高,足够系统的使用。MAX197 的中断输出引脚 INT 连接 AT89C51 的 P1.0 引脚, HBEN 引脚连接 P2.0 引脚,片选 CS 连接 P1.6 引脚。MAX197 转换的输出连接单片机的 P0 端口。

A/D 芯片 MAX197 具有两种电压基准模式,分别为内部 4.096V 电压基准和外部电压基准。本例采用内部电压基准模式,使用 MAX197 的内部时钟作为采集转换的时钟,由外部单片机控制采集。为了给 MAX197 提供稳定的供电,本例中采用通过线性稳压器 MAX8875 来实现 5V 的精确电压输出。

MAX8875 是一款低压差线性稳压芯片,同样是 MAXIM 公司生产的。MAX8875 具有如下所示特点。

- 输入电压的范围为+2.5V~+6.5V。
- 输出电流达到 150mA。
- 提供多种选择的输出电压值: 1.5V、2.5V、2.7V、3.0V、3.3V 和 5V,输出电压的初始精度为 $\pm 1\%$ 。
- 有“POWER—OK”输出,用于当输出电压超出稳压范围时产生报警信号。
- 输出端只需连接 1 μ f 的陶瓷电容,就可确保负载电流高达 150mA 的稳定性。
- 芯片支持过热和短路保护。
- 具有电源反接保护。
- 1 μ A 最大关断电流。

低压差线性稳压芯片 MAX8875 的引脚封装,如图 36-11 所示。MAX8875 的各引脚功能如下所示。

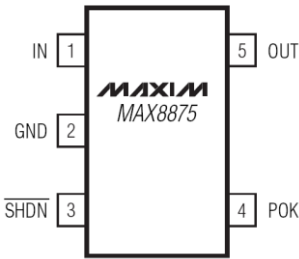


图 36-11 MAX8875 的引脚封装

- IN (Pin1): MAX8875 稳压器输入引脚,输入电压范围为+2.5V~+6.5V。该引脚可以旁路一个 1 μ f 的陶瓷电容接地。
- GND (Pin2): 接地引脚。
- SHDN (Pin3): MAX8875 低功耗模式控制引脚。该引脚低电平有效,此时工作电流可以下降到 1 μ A 以下。
- POK (Pin4): MAX8875 的“POWER—OK”输出脚。当输出电压超出稳压范围时,该引脚便产生报警信号。使用报警功能时,需要将该引脚和 OUT 引脚直接连接一个 100k Ω 的电阻;如果不使用报警功能,则将此引脚悬空即可。
- OUT (Pin5): MAX8875 稳压器输出引脚。不同的型号可输出不同的稳压值: 1.5V、2.5V、2.7V、3.0V、3.3V 和 5.0V。

这里选用固定 5V 输出的 MAX8875EUK50 为 MAX197 提供稳定的供电电压。

36.3.2 建立项目

这里采用 Keil C51 软件，进行程序设计。具体操作步骤如下：

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择 “Project” → “New Project” → “μ Vision Project” 命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89C51，如图 36-12 所示。
- (3) 单击 “确定” 按钮，此时弹出 “μ Vision3” 对话框，如图 36-13 所示。单击 “是” 按钮，完成工程的建立。
- (4) 选择 “File” → “New” 命令，新建一个程序文件，并保存为 *.c 文件。可以在其中输入程序代码。

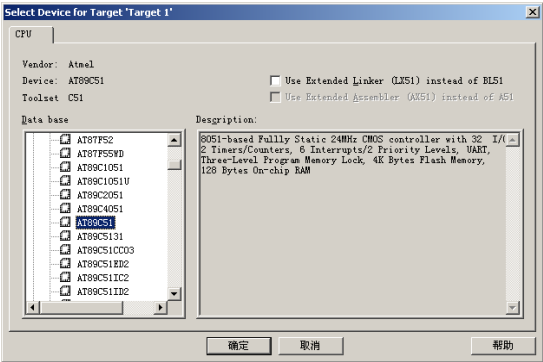


图 36-12 选择单片机 AT89C51

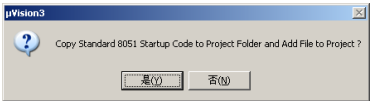


图 36-13 “μ Vision3” 对话框

36.3.3 程序设计

在 Keil μVision3 集成开发环境中，使用 C51 语言进行程序设计。AT89C51 控制 MAX197 实现 A/D 转换和采集的程序代码示例如下：

```
#include <reg51.h> //头文件
#include <absacc.h>
#include <stdio.h>

#define uchar unsigned char //MAX197 控制字节
#define adch0 XBYTE[0x0100]

uchar CHDataL,CHDataH; //转换结果
sbit ADINT = P1^0; //MAX197 中断输出位
sbit HBEN = P2^0; //MAX197 数据总线复用控制位
sbit CS=P1^7; //MAX197 片选信号

void ADRead(void); //AD 转换读取函数声明

void main() //主函数
{
    uchar Select_Menu; //功能选择

    SCON=0x50; //初始化串行口模式 1
    TMOD=0x20; //初始化 T1 为定时功能，模式 2
    PCON=0x80; //设置 SMOD=1
```

51 单片机开发与应用技术详解

```
TL1=0xF4;                //波特率 4800bit/s , 初值
TH1=0xF4;

TR1 = 1;                  //启动 T1
TI = 1;                   //启动发送

while(1)
{
    printf("A/D Converter MAX197 Control!");
    printf (" Select Control Menu Option\n"); //菜单信息
    printf (" 0. Read Channel 0.\n");
    printf (" 1. Read Channel 1.\n");
    printf (" 2. Read Channel 2.\n");
    printf (" 3. Read Channel 3.\n");
    printf (" 4. Read Channel 4.\n");
    printf (" 5. Read Channel 5.\n");
    printf (" 6. Read Channel 6.\n");
    printf (" 7. Read Channel 7.\n");
    printf (" 8. Enable MAX197.\n");
    printf (" 9. EXIT.\n");
    printf (" Please Input Menu Option:\n");

    Select_Menu = _getkey();          //从键盘输入选择数字
    switch(Select_Menu)
    {
        case '0':                    //读取通道 0
            printf ("\n You Select 0. ReadChannel 0.\n");
            adch0=0x40;               //控制字节
            ADRead();                 //AD 转换
            printf("Channel 0:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
            break;
        case '1':                    //读取通道 1
            printf ("\n You Select 1. ReadChannel 1.\n");
            adch0=0x41;               //控制字节
            ADRead();                 //AD 转换
            printf("Channel 1:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
            break;
        case '2':                    //读取通道 2
            printf ("\n You Select 2. ReadChannel 2.\n");
            adch0=0x42;               //控制字节
            ADRead();                 //AD 转换
            printf("Channel 2:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
            break;
        case '3':                    //读取通道 3
            printf ("\n You Select 3. ReadChannel 3.\n");
            adch0=0x43;               //控制字节
            ADRead();                 //AD 转换
            printf("Channel 3:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
            break;
        case '4':                    //读取通道 4
            printf ("\n You Select 4. ReadChannel 4.\n");
            adch0=0x44;               //控制字节
            ADRead();                 //AD 转换
            printf("Channel 4:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
            break;
```



```

case '5':                                //读取通道 5
    printf ("\n You Select 5. ReadChannel 5.\n");
    adch0=0x45;                          //控制字节
    ADRead();                            //AD 转换
    printf("Channel 5:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
    break;
case '6':                                //读取通道 6
    printf ("\n You Select 6. ReadChannel 6.\n");
    adch0=0x46;                          //控制字节
    ADRead();                            //AD 转换
    printf("Channel 6:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
    break;
case '7':                                //读取通道 7
    printf ("\n You Select 7. ReadChannel 7.\n");
    adch0=0x47;                          //控制字节
    ADRead();                            //AD 转换
    printf("Channel 7:CHDataL=%d,ChDataH=%d\n",CHDataL,CHDataH);
    break;
case '8':                                //使能 MAX197
    printf ("\n You Select 8. Enable MAX197.\n");
    CS=0;                                //CS 低电平有效,使能 MAX197
    printf ("\nMAX197 Enabled.\n");
    break;
case '9':                                //退出程序
    printf ("\n You Select 9. Exit\n");
    goto Exit;                            //转向 Exit 标号处
    break;
default:
    printf ("\n Error: Please Select Right Menu Option\n");
    break;
};
}
Exit: printf("Exit the program!");        //退出
}

```

在该程序中,首先初始化串口,然后通过扫描键盘的输入来执行 A/D 转换程序,实现 0~7 共 8 个通道的模拟电压的转换和采集。A/D 转换的结果将通过串口打印输出。

在主程序中,调用了 A/D 转换子函数 ADRead。在调用该子函数之前,首先写 MAX197 的控制字 adch0,用于指定测量的通道。其中,PD1=0,PD0=1,表示正常工作,采用内部时钟模式;ACQMOD=0 表示内部控制采集;RNG=0,BIP=0 表示采用 0~5V 测量范围;A0~A2 用来设置测量的模拟通道数。

A/D 转换子函数 ADRead 的程序代码示例如下:

```

void ADRead(void)
{
    while (ADINT!=0)                    //查询 MAX197 的中断输出,判断是否完成转换
    {
        HBEN = 0;                      //首先置 HBEN = 0,即先读低位

        CHDataL = P0;                  //读取 A/D 转换结果的低位

        HBEN = 1;                      //设置 HBEN = 1,再读高位
        CHDataH = P0;                  //读取 A/D 转换结果的高位
    }
}

```

```
HBEN = 0; //重置 HBEN = 0
}
```

程序输入完毕后，便可以程序编译。MAX197 是一个非常好用的 A/D 转换芯片。对于其他一些 A/D 转换芯片，使用方法基本类似，可以参考相应的数据手册来了解其用法。

36.3.4 程序仿真

当程序编译通过后，便可以利用 Keil μ Vision3 集成开发环境提供的信号仿真功能，来对程序进行仿真分析。具体操作步骤如下：

- (1) 在 Keil μ Vision3 集成开发环境中，选择“Debug”→“Start/Stop Debug Session”命令，进入仿真分析模式。
- (2) 选择“View”→“Serial Window”→“UART #0”命令，打开串口仿真接口。
- (3) 选择“Peripherals”→“I/O-Ports”→“Port 0”命令，打开并行端口 P0 仿真界面。按照同样的方法，打开并行端口 P1 的仿真界面。
- (4) 选择“Debug”→“Go”命令，开始执行仿真。
- (5) 此时，在串行仿真接口中便输出读写 MAX197 的控制菜单。首先输入“8”使能 MAX197，如图 36-14 所示。

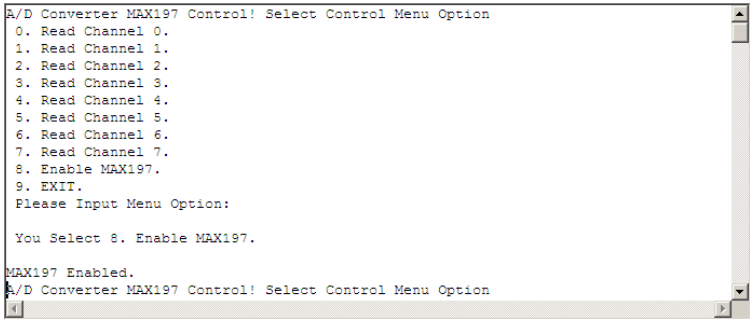


图 36-14 使能 MAX197

- (6) 接着输入“0”，模拟采集通道 0。此时程序处于等待状态，等待 MAX197 输出 A/D 转换结束的中断信号。该信号通过 P1.0 引脚输入。
- (7) 在 P1 端口的仿真界面上置 P1.0 引脚为低电平，即模拟 MAX197 转换结束的中断输出。
- (8) 此时程序便读入 P0 端口的数据，作为 MAX197 的转换输出。
- (9) 当仿真操作完毕后，选择“Debug”→“Stop Running”命令，便可以结束程序的仿真运行。
- (10) 选择“Debug”→“Start/Stop Debug Session”命令，可以退出程序仿真调试环境。当仿真无误后，便可以将程序下载到单片机中执行。

36.4 小结

本章首先详细介绍了 A/D 转换的原理、A/D 转换器的技术参数及 A/D 转换器的选用原则。接着，本章介绍了一个高性能的 8 通道 A/D 转换器 MAX197。其中，详细讲解了 MAX197 的特性、引脚功能，以及接口、控制和时序逻辑等。最后，本章通过一个完整的实例介绍了 A/D 转换器 MAX197 与单片机的接口。A/D 转换在测控领域，特别是模拟信号的数据采集系统中有着广泛的应用，读者应熟练掌握 A/D 转换的相关知识及 A/D 转换器的使用。

第 37 章 单片机读写智能 IC 卡

智能 IC (Intelligent Card) 卡, 是指一张集成了部分电路的卡片。其一般采用固定大小, 其内部的电路主要用于存储和处理数据。智能 IC 卡一般简称为智能卡或 IC 卡。ATM 取款机、公交卡及学校的饭卡均是智能 IC 卡。IC 卡带来了灵活方便的身份识别和支付等功能, 目前已广泛应用于交通、校园、金融、电信、医疗及门禁系统等领域。

本章将主要介绍智能 IC 的概述, 常用的智能 IC 卡芯片及其读写方式。本章还通过一个实例, 介绍了 51 系列单片机如何读写智能 IC 卡。

37.1 智能 IC 卡概述

从本质上说, 智能 IC 卡是集成电路的组合。IC 卡制作成标准大小的卡片形式, 是为了方便携带及便于读写操作。智能 IC 卡目前已经深入到人们生活的方方面面。下面介绍 IC 卡的分类, 以及常见的接触式 IC 卡和非接触式 IC 卡。

37.1.1 智能 IC 卡分类

智能 IC 卡按照用途来分, 可以分为如下几类。

- 身份识别, 例如门禁卡、居民身份证等。
- 支付工具, 例如 ATM 取款机、公交卡、学校的饭卡等。
- 加密/解密, 例如数据备份卡、加密 IC 卡等。
- 信息, 例如手机的 SIM 卡等。

智能 IC 卡按照使用的方式来分, 可以分为接触式 IC 卡和非接触式 IC 卡。

- 接触式 IC 卡通过机械触点从 IC 卡读写设备获取能量和交换数据;
- 非接触式 IC 卡通过射频感应从读写设备获取能量和交换数据。非接触式 IC 卡又称为射频卡。

智能卡 IC 卡按所集成的芯片类型的不同, 可分为如下所示的三类。

- 存储器 IC 卡: 卡内的集成电路是可用电擦除的可编程只读存储器 EEPROM, 其仅具有数据存储功能, 没有数据处理能力。存储卡本身无硬件加密功能, 只在文件上加密, 很容易被破解。
- 逻辑加密 IC 卡: 卡内的集成电路包括加密逻辑电路和可编程只读存储器 EEPROM。加密逻辑电路可在一定程度上保护卡和卡中数据的安全性。
- 智能卡 (CPU 卡): 卡内的集成电路包括微处理器 CPU、可编程只读存储器 EEPROM、随机存储器 RAM 和固化在只读存储器 ROM 中的卡内操作系统 COS (Chip Operating System)。依靠卡中的操作系统, 智能卡可以完成更多的任务, 并且可以确保卡中数据的安全可靠。

37.1.2 接触式 IC 卡

接触式 IC 卡是在卡片上引出了机械触点的 IC 卡。接触式 IC 卡内部的芯片通过机械触点来进行数据读写操作。使用时, 将 IC 卡插入读写器的插槽内, 由读写器完成对芯片的读出和

写入操作。

接触式 IC 卡每次读写时，都必须把 IC 卡正确插入到读写器中才能完成数据的交换。因此，存在着操作不方便、触点易磨损、读写速度慢等缺点。但是接触式 IC 卡制作起来比较简单，成本也比较低，目前使用的非常广泛。本章就以接触式 IC 卡为主进行介绍。

Atmel 公司的 IC 卡芯片是目前市场上最常见的。其提供了存储器型、逻辑加密型等多种 IC 卡类型，可广泛应用于多种场合。

37.1.3 非接触式 IC 卡

非接触式 IC 卡没有明显的触点，其内部除了存储单元、控制逻辑外，还集成有感应线圈。非接触式 IC 卡通过线圈射频感应从读写设备获取能量和交换数据。在使用时，将非接触式 IC 卡放在读写器附近一定的距离之内，读写器通过射频为 IC 卡内部的电路提供能量，并通过一定的协议完成数据的交换。

非接触式 IC 卡由于不存在机械触点，因此使用方便快捷、不易损坏。非接触式 IC 卡制作比较复杂，成本也比较高，其常用于使用频繁、可靠性要求高的场合。目前，很多应用领域的智能 IC 卡都开始使用非接触式 IC 卡。

Philips 公司的 Mifare 技术是当今非接触式智能 IC 射频卡的主流技术，目前已经被指定为非接触式 IC 卡的国际标准（ISO/IEC 14443A 标准）。现在一些大型的 IC 卡制造商、IC 卡读写器制造商及相关软件设计公司等都以 Philips 公司的 Mifare 技术作为标准。

37.2 智能 IC 卡芯片

下面以接触式智能 IC 为例进行介绍。最常见的接触式智能 IC 卡，其内部芯片采用 Atmel 公司的 AT45DB041。这里便以该芯片为例，介绍 IC 卡的操作。

37.2.1 IC 卡芯片 AT45DB041 简介

AT45DB041 是 Atmel 公司生产的 4Mbit 串行数据 Flash，支持 ISP 在系统重编程（擦写和编程），可用于语音、图像和数据的存储，特别适合于存储器型 IC 的设计。AT45DB041D 是最新的型号，其主要功能特性如下：

- 单 2.5V 供电或 2.7V~3.6V 单电源供电。
- 串行接口结构，兼容 SPI 串行外设接口。
- 支持页编程操作，可在单周期完成编程，2048 页（264 字节/页）主内存。
- 可进行页擦除或块擦除。
- 具有两个 SRAM 数据缓冲器，可以允许在系统重编程时接收数据。
- 可以连续进行读操作。
- 具有内部程序和控制定时器。
- 具有硬件数据保护功能。
- 低功耗，输入输出兼容 CMOS 和 TTL 电平。
- 最大 66MHz 的工作频率。
- 100000 次擦写次数。
- 数据可以保存 20 年以上。

芯片 AT45DB041D 的引脚封装，如图 37-1 所示。芯片 AT45DB041D 的各引脚功能如下：

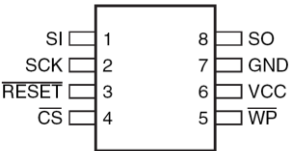


图 37-1 AT45DB041D 的引脚封装

- SI (Pin1): 串行输入脚。
- SCK (Pin2): 串行时钟脚。
- $\overline{\text{RESET}}$ (Pin3): 复位脚, 低电平有效。
- $\overline{\text{CS}}$ (Pin4): 片选脚, 低电平有效。
- $\overline{\text{WP}}$ (Pin5): 硬件页写操作保护脚, 低电平有效, 低电平时主内存空间的前 256 页不允许进行编程操作。
- VCC (Pin6): 电源引脚。
- GND (Pin7): 接地引脚。
- SO (Pin8): 串行输出脚。

AT45DB041D 使用串行数据接口, 该接口与 SPI 串行接口兼容。相比并行总线接口, 采用串行数据接口可以减少 I/O 引脚数目、减少机械触点、缩小芯片封装大小和增加系统的可靠性。SPI 接口是一种三线制的全双工穿行同步数据传输接口, 三根信号线分别是串行输入信号 SI、串行输出信号 SO 和串行时钟信号 SCK。

IC 卡芯片 AT45DB041D 具有 4Mbit 的内存空间。其内存空间由 2048 页构成, 每页 264 个字节。另外, AT45DB041D 还有两个 SRAM 数据缓冲区, 每个缓冲区的大小也是 264 个字节。SRAM 缓冲区的作用是使主内存编程的时候允许接收新的数据。AT45DB041D 内部结构如图 37-2 所示。

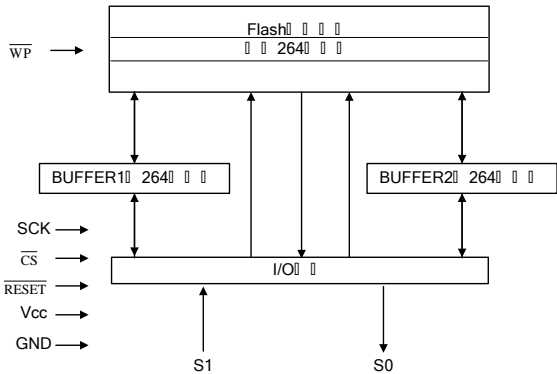


图 37-2 IC 卡芯片 AT45DB041D 内部结构框图

37.2.2 AT45DB041D 的内存空间及其读写

- AT45DB041D 共有 4Mbit 的内存空间, 其结构按照页面、块和扇区来组织, 分别介绍如下:
- 以页面为单元, AT45DB041D 内存中共分为 2046 页, 每页共 264 个字节。
 - 以块为单元, AT45DB041D 内存中共分为 256 块, 每块共 2112 个字节, 8 页构成一块。
 - 以扇区为单元, AT45DB041D 内存中共分为 6 个扇区, 扇区 0 由 8 页构成, 相当于第 0 块; 扇区 1 由 248 页构成, 对应第 1 块到第 31 块; 扇区 2 由 256 页构成, 对应第 32 块到第 63 块; 扇区 3、4、5 均为 512 页构成, 每个扇区对应 64 块的空间。

在对 AT45DB041D 的内存空间进行操作时, 所有编程写操作都以页面为基本单位, 而擦除操作可以选择页面操作或块操作。

对 AT45DB041D 的操作实际上就是对内部内存空间的操作。由于 AT45DB041D 内部除主内存空间外, 还有两个 264B 的 BUFFER。因此, 在读写时可以直接对主内存空间操作, 也可以以 BUFFER 作为缓冲器来操作。AT45DB041D 的读流程和写流程, 分别如图 37-3 和图 37-4 所示。

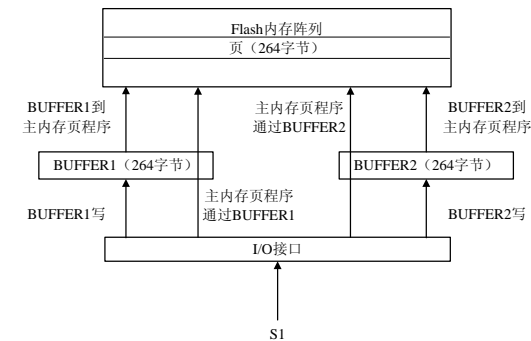


图 37-3 AT45DB041D 的读流程示意图

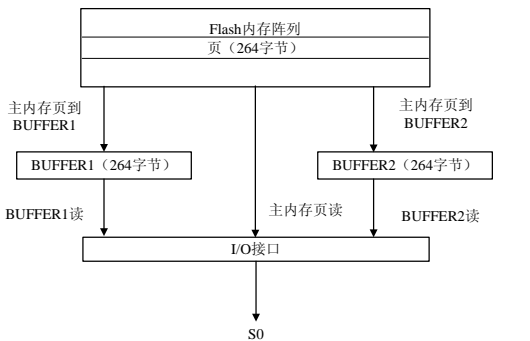


图 37-4 AT45DB041D 的写流程示意图

37.2.3 AT45DB041D 的指令

对 IC 卡芯片 AT45DB041D 的读写操作由主控制器发送指令来实现。由于 AT45DB041D 采用串行接口，指令执行时，首先 CS 为低电平，然后启动 SCK 时钟控制 SI 输入脚，并装载指令代码和操作对象的地址，完成一条指令的输入。

其中，操作对象的寻址主要包括 4Mbit 的主内存寻址和 BUFFER 寻址。BUFFER 寻址通过 BFA8~BFA0 共 9 位地址来实现，可以定位 BUFFER 内具体字节的地址。AT45DB041D 的 4Mbit 主内存区寻址通过 PA10~PA0 和 BA8~BA0 来实现的。其中，PA10~PA0 共 11 位地址，用于确定页面的位置；BA8~BA0 共 9 位地址，用于确定页面内具体字节的位置。

IC 卡芯片 AT45DB041D 提供多种读写指令，由于串行接口 SPI 工作模式的不同，其具体的指令代码有所区别。这里以 SPI 工作模式 3 为例，介绍常用的几个读写指令。

1. 状态寄存器读指令

状态寄存器读指令用来读取状态寄存器所指示的当前 IC 卡状态。状态寄存器读指令的指令代码为 D7H。在 SI 端口串行输入该指令码后，SO 端口将在紧接着的 8 个时钟周期，按照高位在前的格式串行输出状态寄存器的 8 位。AT45DB041D 状态寄存器的字节内容，如图 37-5 所示。

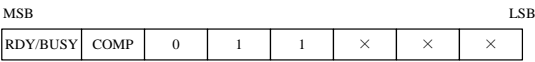


图 37-5 AT45DB041D 的状态寄存器

其中，状态寄存器字节的高 5 位包含了 AT45DB041D 芯片状态信息，低 3 位保留不使用。其中各位的含义如下：

- RDY/BUSY：芯片忙状态标志。如果该位为 1，则芯片处于不忙的状态，可以接收下一条指令；如果该位为 0，则芯片处于忙的状态。
- COMP：主内存页与缓冲区比较操作结果。

状态寄存器的内容在 SO 引脚将会循环输出。在输出过程中，数据如果有变化，将会实时更新。

2. 主内存页读取指令

主内存页读取指令用于从 2048 页中选择任何一页读取数据，其指令代码为 D2H。该指令无须通过两个数据 SRAM 缓冲区。该指令执行时，首先向 AT45DB041D 芯片串行输入指令码 D2H，紧接着输入 24 位地址位和 32 位无关位，便可以启动一次主内存页读取。

其中，24 位地址位的前 4 位为保留位，接下来的 11 位定义为页面地址 PA10~PA0，最后 9 位定义为页面内起始字节地址 BA8~BA0。主内存页读取同样是循环的，即如果 CS 引脚和时钟 SCK 持续有效，一页读完后重新从该页开头继续读取，CS 引脚由低变高将终止读操作。

3. 写缓冲区指令

写缓冲区指令用于将数据从外部控制器移位输出到 AT45DB041D 芯片的内部 BUFFER 中，对应 BUFFER1 的指令代码为 84H，对应 BUFFER2 的指令代码为 87H。该指令执行时，首先向 AT45DB041D 芯片串行输入指令码，紧接着输入 15 位无关位和 9 位地址位，然后是写入的数据，便可以启动写缓冲区指令。

其中，9 位地址位 BFA8~BFA0 用于定义要写入的 BUFFER 的第一个字节的位置。写缓冲区同样是循环的，即如果 $\overline{\text{CS}}$ 引脚和时钟 SCK 持续有效，数据写到缓冲区的末尾后，重新从头开始继续写入， $\overline{\text{CS}}$ 引脚由低变高将终止写缓冲区操作。

4. 由缓冲区写主内存指令

由缓冲区写主内存指令用来将写入 BUFFER 的数据写入主内存，对应 BUFFER1 的指令码为 88H，对应 BUFFER2 的指令码为 89H。该指令执行时，首先向 AT45DB041D 芯片串行输入指令码，紧接着输入 4 位保留位、11 位地址位和 9 位无关位，便可以启动由缓冲区写主内存指令。

该指令在使用前，必须先删除主内存中被写的页面。当 $\overline{\text{CS}}$ 引脚由低变高，存储在缓冲区中的数据将传送到 AT45DB041D 主内存的特定页面中。

37.3 单片机读写智能 IC 卡实例

采用 AT45DB041D 芯片的 IC 卡本质上属于存储器件，可以通过其 SPI 串行接口来对其进行读写操作。在实际使用时，一般都规定好信息的特殊存储格式，以满足不同场合的需要。这里为了方便，只介绍如何使用单片机读写 IC 卡芯片 AT45DB041D，用户可以根据需要自行定义数据信息的格式。

由于 IC 卡芯片 AT45DB041D 采用串行 SPI 接口，因此需要选择带有 SPI 总线接口的单片机。标准的 8051 单片机没有集成 SPI 总线控制器，这里选择 Atmel 公司带有 SPI 接口的 AT89S8253 单片机。

37.3.1 电路图

采用 Atmel 公司的 AT89S8253 单片机来读写 IC 卡的电路原理图，如图 37-6 所示。电路中所使用的元器件列表，如表 37-1 所示。

表 37-1 元器件

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89S8253	1 个	R2	200 Ω	1 个
U2	AT45DB041D	1 个	R3	10k Ω 排阻	1 个
C1、C2	30pf	2 个	R4	4.7k Ω	1 个
C3、C4	0.1 μ F	2 个	Reset	按键开关	1 个
C10	10 μ F 电解电容	1 个	Y1	11.0592MHz	1 个
R1	5k Ω	1 个			

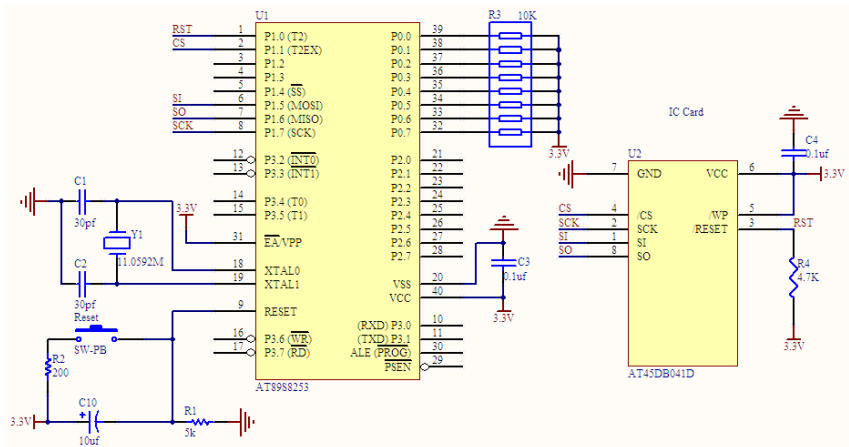


图 37-6 电路原理图

其中，AT45DB041D 的片选信号 \overline{CS} 连接单片机的 P1.1 引脚， \overline{RESET} 引脚连接 P1.0 引脚， \overline{WP} 引脚连接高电平。AT45DB041D 的串行接口 SCK、SI 和 SO 引脚分别连接 P1.7、P1.5 和 P1.6 引脚。

37.3.2 SPI 接口单片机 AT89S8253 简介

AT89S8253 是 Atmel 公司生产的 8 位微处理器，它与 51 单片机完全兼容，但在 P1 端口 (P1.4 ~P1.7) 上扩展了 SPI 接口。其主要功能特性如下：

- 2.7V~5.5V 工作电压。
- 12KB 的在系统重编程可下载 Flash 和 256B 的内部 RAM。
- 2KB 的 EEPROM 存储器。
- 32 位和 8051 兼容的可编程 I/O 线。
- 3 个 16 位定时/计数器和 9 个中断源。
- SPI 串行接口。
- 可编程看门狗定时器。

AT89S8253 中和 SPI 串行数据传输相关的各引脚功能如下：

- \overline{SS} (P1.4)：从器件使能引脚，低电平有效。
- MOSI (P1.5)：主机数据输出从机数据输入引脚。
- MISO (P1.6)：主机数据输入从机数据输出引脚。
- SCK (P1.7)：主机时钟输出从机时钟输入引脚。

这里需要使用单片机 AT89S8253 的 SPI 接口来控制 IC 卡芯片 AT45DB041D，因此首先需要了解其内部的 SPI 寄存器的功能和使用方法。AT89S8253 共有三个与 SPI 相关的寄存器，即控制寄存器 SPCR、状态寄存器 SPSR 和数据寄存器 SPDR。下面分别进行介绍。

控制寄存器 SPCR 的地址为 D5H，其字节内容，如图 37-7 所示：

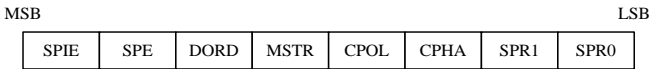


图 37-7 控制寄存器的字节内容

其中，控制寄存器各位的含义如下所示。

- SPIE：SPI 串行中断使能位。SPIE =0 时，禁止 SPI 串行中断；SPIE =1 时，若 EA 也为 1，则使能 SPI 串行中断。
- SPE：SPI 串行通道使能位。SPE =0 时，禁止 SPI 串行通道；SPE =1 时，使能 SPI 串行通道。

- **DORD**: 数据顺序设置位。DORD=0 时，数据串行传输时低位在前；DORD =1 时，数据串行传输时高位在前。
- **MSTR**: SPI 主从模式设置位。MSTR =0 时，选择从模式；MSTR =1 时，选择主模式。
- **CPOL**: 时钟极性设置位。CPOL =0 时，主器件时钟 SCK 在空闲时为低电平；CPOL =1 时，主器件时钟 SCK 在空闲时为高电平。
- **CPHA**: 时钟相位设置位。CPHA 和 CPOL 一起控制主从器件之间的时钟和数据的相位关系。
- **SPR1 和 SPR0**: 时钟频率设置位。SPR1 和 SPR0 主要用于主模式器件的 SCK 频率设置，对于从模式器件没有作用。设置为 00 时，SCK 频率为 FOSC/4；设置为 01 时，SCK 频率为 FOSC/16；设置为 10 时，SCK 频率为 FOSC/64；设置为 11 时，SCK 频率为 FOSC/128。这里的 FOSC 为单片机外接晶振的频率。

状态寄存器 SPSR 的地址为 AAH，其字节内容如图 37-8 所示。



图 37-8 状态寄存器的字节内容

其中，控制寄存器各位的含义如下所示。

- **SPIF**: SPI 串行中断标志位。当一次 SPI 串行传输完成后，SPIF 被置为 1，如果 SPIE 和 EA 均置为 1，则会产生 SPI 串行中断。读 SPI 状态寄存器 SPSR 将自动清除该位。
- **WCOL**: 写冲突标志位。在 SPI 串行数据传输过程中，写数据寄存器 SPDR 时，WCOL 将置 1。在 SPI 串行数据传输过程中，读数据寄存器 SPDR 值会发生错误。读 SPI 状态寄存器 SPSR 将自动清除该位。

数据寄存器 SPDR 的地址为 86H，其字节内容如图 37-9 所示。



图 37-9 数据寄存器的字节内容

其中，SPD0~SPD7 为 8 位串行数据。

37.3.3 建立项目

这里采用 Keil C51 软件，进行程序设计。具体操作步骤如下：

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New Project”→“ μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S8253，如图 37-10 所示。
- (3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 37-11 所示。单击“是”按钮，完成工程的建立。
- (4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.c 文件，可以在其中输入程序代码。

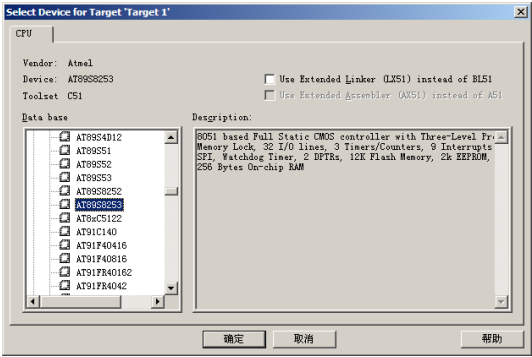


图 37-10 选择单片机 AT89S8253

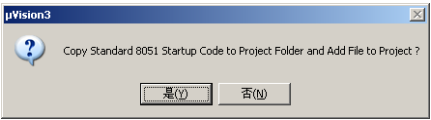


图 37-11 “ μ Vision3”对话框

37.3.4 IC 卡芯片 AT45DB041D 读写子函数

对 IC 卡芯片 AT45DB041D 的读写需要使用相关的指令，并满足一定的时序。这里将用到的函数封装成子函数的形式，方便调用。

1. 延时函数

延时函数 Delay 用于延时指定的时间，用来构成读写 IC 卡芯片所需要的时序。在程序中通过一个空循环语句便可以实现延时。延时函数 Delay 的程序代码示例如下：

```
void Delay(uint t)
{
    uint i,j;
    for(j=0;j<t;j--)
    {
        for (i=0;i<125;i++)           //空循环
        {
        }
    }
}
```

2. 写数据寄存器函数

写数据寄存器函数 SPIWrite 用于向 AT89S8253 的数据寄存器 SPDR 写数据，该数据将通过 SPI 口串行输出至 IC 卡芯片 AT45DB041D。程序中通过读寄存器 SPSR 来判断传输完成。写数据寄存器函数 SPIWrite 的程序代码示例如下：

```
void SPIWrite(uchar dat)
{
    SPDR = dat;           //写数据寄存器 SPDR
    while (!(SPSR & 0x80)) //读 SPSR，等待传输完成
    {
    }
}
```

3. 读 IC 卡状态函数

读 IC 卡状态函数 ICStatus 用来读取 IC 卡芯片 AT45DB041D 的状态。程序中首先使能 IC 卡芯片，然后通过 SPIWrite 函数写入读 IC 卡芯片状态指令。IC 卡芯片返回的状态数据将保存在数据寄存器 SPDR 中。读 IC 卡状态函数 ICStatus 的程序代码示例如下：

```
uchar ICStatus(void)
{
    P1_1 = 0;           //使能 IC 卡芯片

    SPIWrite(StatusRead); //向 IC 卡芯片写入读 IC 卡芯片状态指令
    SPIWrite(0x00);       //写无关位

    P1_1 = 1;           //禁用 IC 卡芯片
    return SPDR;         //返回 IC 卡芯片状态
}
```

4. 写 IC 卡芯片函数

写 IC 卡芯片函数 WriteICFun 用于将数据写入 IC 卡芯片 AT45DB041D 的 BUFFER 中。在程序中，首先通过 ICStatus 函数检查 IC 卡芯片的状态，如果忙则等待。如果 IC 卡芯片不忙，则执行写 BUFFER1 指令。如果 BUFFER1 已满，则将 BUFFER1 中的数据转移到芯片的主内存页中。写 IC 卡芯片函数 WriteICFun 的程序代码示例如下：

```
void WriteICFun(uchar dat)
{
```

```

uchar status;

status= ICStatus();           //检查 IC 卡芯片的状态
while ((status&0x80)==0x00)   //如果 IC 卡芯片忙，则等待
{
}

Pl_1 = 0;                     //使能 IC 卡芯片
SPIWrite(BUFFER1Write);       //IC 卡芯片的写 BUFFER1 指令代码
SPIWrite(0x00);               //写入 8 位无关位
SPIWrite((uchar) (BufStartAddr>>8)); //写入 7 位无关位及 9 位 BUFFER 起始字节地址的第 1 位
SPIWrite((uchar) BufStartAddr); //写入 9 位 BUFFER 起始字节地址的后 8 位
SPIWrite(dat);                //写入数据
Pl_1 = 1;                     //禁用 IC 卡芯片

BufStartAddr++;               //指向下一 BUFFER 起始字节地址

if (BufStartAddr > 263)       //如果 BUFFER 已满
{
    BufStartAddr = 0;         // buffer 起始字节地址重置 0
    if (PageAddr < 2047)      //如果 IC 卡芯片的主内存页不满
    {
        /* buffer 数据写入主内存页 */
        Pl_1 = 0;            //使能 IC 卡芯片
        SPIWrite(BUFFER1ToMain); //由缓冲区写主内存指令代码
        SPIWrite((uchar) (PageAddr>>7)); //写入 4 位保留位加上 11 位页地址的高 4 位
        SPIWrite((uchar) (PageAddr<<1)); //写入 11 位页地址的低 7 位和 1 位无关位
        SPIWrite(0x00);       //写入 8 位无关位
        Pl_1 = 1;            //禁用 IC 卡芯片

        PageAddr++;           //指向下一页地址
    }
}
}

```

5. 读 IC 卡芯片函数

读 IC 卡芯片函数 `ReadICFun` 用来读取 IC 卡芯片中保存的数据。在程序中，首先通过 `ICStatus` 函数检查 IC 卡芯片的状态，如果忙则等待。如果 IC 卡芯片不忙，则执行主内存页读指令。当一页已经读完的时候，便继续读下一页。读 IC 卡芯片函数 `ReadICFun` 的程序代码示例如下：

```

uchar ReadICFun()
{
    uchar status;
    uchar tmp;

    status= ICStatus();           //检查 IC 卡芯片的状态
    while ((status&0x80)==0x00)   //如果 IC 卡芯片忙，则等待
    {
    }

    Pl_1 = 0;                     //使能 IC 卡芯片

```

```
SPIWrite(MainPageRead) ; //写入主内存页读指令代码
tmp = (uchar) (PageAddr>>7); //11 位页地址的高 4 位
SPIWrite(tmp); //写入 4 位保留位及 11 位页地址的高 4 位
tmp = (uchar) (PageAddr<<1) | ((uchar) (PageStartAddr>>8) & 0x01);
SPIWrite(tmp); //写入页地址的低 7 位及 9 位页起始字节地址的最高位
tmp = (uchar) (PageStartAddr);
SPIWrite(tmp); //写入 9 位页起始字节地址的低 8 位
SPIWrite(0x00) ; //写入 8 位无关位
SPIWrite(0x00) ; //写入 8 位无关位
SPIWrite(0x00) ; //写入 8 位无关位
SPIWrite(0x00) ; //再写入 8 位无关位, 共写入 32 位无关位
SPIWrite(0xff) ; //写入任意值, 用来读 IC 卡芯片返回的数据
Pl_1 = 1; //禁用 IC 卡芯片

PageStartAddr++; //指向下一页中起始字节地址

if (PageStartAddr > 263) //如果读完一页
{
    PageStartAddr = 0; //页起始字节地址重置 0
    if (PageAddr < 2047) //如果主内存页没有读完
        PageAddr++; //指向下一页地址
}

return SPDR; //返回数据
}
```

37.3.5 主程序

这里使用 AT89S8253 单片机来读写 IC 卡芯片 AT45DB041D。主函数中首先初始化串行口为模式 1，波特率为 4800bit/s。接着通过 while 循环语句来扫描串口输入，根据输入数据来调用函数执行不同的功能。主函数的程序代码示例如下：

```
#include <REG8253.H> //头文件
#include <stdio.h>

#define uchar unsigned char
#define uint unsigned int

#define BUFFER1Write 0x84 // BUFFER1 写指令代码
#define BUFFER1ToMain 0x88 //由缓冲区写主内存指令代码
#define MainPageRead 0xD2 //主内存页读指令代码
#define StatusRead 0xD7 //状态寄存器读指令代码

#define WriteICMaxLen 8 //写 IC 卡芯片最大长度
#define ReadICMaxLen 8 //读 IC 卡芯片最大长度

uint PageStartAddr; //页中起始字节地址
uint PageAddr; //页地址, 16 位中低 9 位为有效位
uint BufStartAddr; // BUFFER 中起始字节地址, 16 位中低 11 位为有效位
uchar WriteIC[WriteICMaxLen]; //要写入 IC 卡的数据
```

第 37 章 单片机读写智能 IC 卡

```
uchar ReadIC[ReadICMaxLen];           //要从 IC 卡中读出的数据

void Delay(uint t);                    //延时函数
void SPIWrite(uchar dat);              //写数据寄存器函数
uchar ICStatus(void);                 //读 IC 卡状态函数
void WriteICFun(uchar dat);           //写 IC 卡芯片函数
uchar ReadICFun();                    //读 IC 卡芯片函数

void main()                            //主函数
{
    uchar i;
    uchar status;
    uchar Select_Menu;                //功能选择

    SCON=0x50;                        //初始化串行口模式 1
    TMOD=0x20;                        //初始化 T1 为定时功能, 模式 2
    PCON=0x80;                        //设置 SMOD=1
    TL1=0xF4;                         //波特率 4800bit/s, 初值
    TH1=0xF4;

    TR1 = 1;                          //启动 T1
    TI = 1;                           //启动发送

    P1_0 = 1;
    SPCR=0x5d;                        //SPI 控制寄存器

    for (i=0;i<8;i++)                //产生数据, 用于写 IC 卡
    {
        WriteIC[i]=i;
    }

    while(1)
    {
        printf("A/D Converter MAX197 Control!");
        printf (" Select Control Menu Option\n"); //菜单信息
        printf (" 0. Write Data to IC Card.\n");
        printf (" 1. Read Data from IC Card.\n");
        printf (" 2. Read status of IC Card.\n");
        printf (" 3. Reset IC Card.\n");
        printf (" 4. EXIT.\n");
        printf (" Please Input Menu Option:\n");

        Select_Menu = _getkey();      //从键盘输入选择数字
        switch(Select_Menu)
        {
            case '0':                 //写 IC 卡
                printf ("\n You Select 0. Write Data to IC Card.\n");

                BufStartAddr = 0;      // BUFFER 中起始字节地址
                PageStartAddr = 0;     //页起始地址
                PageAddr = 0;          //页地址

                for (i=0;i<WriteICMaxLen;i++) //将数组 WriteIC 写入 IC 卡
                {
```

51 单片机开发与应用技术详解

```
        WriteICFun(WriteIC[i]);
        Delay(2);                //延时
    }
    printf("Data have write into IC Card.\n");
    break;
case '1':                        //读 IC 卡
    printf ("\n You Select 1. Read Data from IC Card.\n");

    BufStartAddr = 0;            // BUFFER 中起始字节地址
    PageStartAddr = 0;           //页起始地址
    PageAddr = 0;                //页地址

    for (i=0;i<ReadICMaxLen;i++) //循环读 IC 卡
    {
        ReadIC[i] = ReadICFun();
        Delay(2);                //延时
    }
    printf("Data have read from IC Card.\n");
    break;
case '2':                        //读 IC 卡状态
    printf ("\n You Select 2. Read status of IC Card.\n");

    status=ICStatus();           //读 IC 卡状态
    printf("IC Card Status=%x\n",status);
    break;
case '3':                        //复位 IC 卡
    printf ("\n You Select 3. Reset IC Card.\n");
    P1_0=0;                      //复位
    Delay(5);
    P1_0 = 1;
    SPCR=0x5d;                   //重置 SPI 控制寄存器
    printf("IC Card Reset!\n");
    break;
case '4':                        //退出程序
    printf ("\n You Select 4. Exit\n");
    goto Exit;                   //转向 Exit 标号处
    break;
default:
    printf ("\n Error: Please Select Right Menu Option\n");
    break;
};

Exit: printf("Exit the program!"); //退出
}
```

通过前面的 IC 卡芯片读写子函数以及这里的主函数，便可以实现 AT89S8253 单片机对 IC 卡芯片 AT45DB041D 的读写操作。当程序通过编译后，便可以在实际硬件中执行。

37.4 小结

本章主要介绍了智能 IC 卡的相关知识。其中对目前市场上广泛使用的 AT45DB041D 接触式 IC 卡芯片进行了详细介绍，包括芯片的功能、内存空间和指令。本章还通过一个具体的实例，介绍了如何使用单片机对 IC 卡芯片进行读写操作。由于 IC 卡芯片采用 SPI 串行数据接口，这里使用了带有 SPI 接口的 AT89S8253 单片机进行读写操作。智能 IC 卡目前得到广泛的使用，读者应该熟练掌握本章内容。

第 38 章 单片机智能锂电池

充电管理

随着微电子技术的发展,各种小型化的便携式设备日益增多,例如手机、数码相机、笔记本等。为了能够更加有效地使用这些电子产品,可充电电池得到快速发展。常见的可充电电池包括镍氢电池、镍镉电池、锂电池和聚合物电池等。其中,锂电池以其高的能量密度、无记忆性和使用寿命长等优点得到广泛的应用。目前绝大部分的手机、数码相机等均使用锂电池。

锂电池对充电器的要求比较高,为了有效地控制锂电池的充电,需要能够对其充电过程进行密切的监控。目前,一般使用单片机配合一定的充电管理芯片来实现锂电池充电的智能管理。本章将主要介绍锂电池及锂电池智能充电器,并通过一款高性能的锂电池充电管理芯片,来介绍如何实现充电的智能化。

38.1 锂电池及其充电概述

锂离子电池以其特有的性能优势已在便携式电器如笔记本、数码相机、摄像机、手机中得到普遍应用。下面首先介绍锂电池及其智能充电的要求。

38.1.1 锂电池概述

锂电池和锂离子电池是 20 世纪开发成功的新型高能电池。它是一类由锂金属或锂合金为负极材料、使用非水电解质溶液的电池。锂电池的正极可以采用 MnO_2 , SOCl_2 , $(\text{CFx})_n$ 等。

最早出现的锂电池来自于发明家爱迪生。由于锂金属的化学特性非常活泼,使得锂金属的加工、保存、使用,对环境要求非常高。所以,锂电池长期没有得到应用。1992 年 Sony 公司成功开发锂离子电池。它的实用化,使人们的手机、笔记本电脑等便携式电子设备重量和体积大大减小,使用时间大大延长。由于锂离子电池中不含有重金属铬,与镍铬电池相比,大大减少了对环境的污染。

锂离子电池目前由液态锂离子电池(LIB)和聚合物锂离子电池(PLB)两类。其中,液态锂离子电池是指 Li^+ 嵌入化合物为正、负极的二次电池。正极采用锂化合物 LiCoO_2 或 LiMn_2O_4 , 负极采用锂-碳层间化合物。锂离子电池由于工作电压高、体积小、质量轻、能量高、无记忆效应、无污染、自放电小、循环寿命长,是 21 世纪发展的理想能源。

锂电池以及锂离子电池的主要特点如下:

- 高能量密度,锂离子电池的重量是相同容量的镍镉或镍氢电池的一半,体积是镍镉的 40%~50%,镍氢的 20%~30%。因此,锂电池具有更高的重量能量比、体积能量比。
- 高电压,单节锂电池电压平均为 3.6V,等于三只镍镉或镍氢充电电池的串联电压。
- 自放电小,可长时间存放。
- 无记忆效应,锂电池不存在镍镉电池的所谓记忆效应,所以锂电池充电前无须放电。
- 寿命长,正常工作条件下,锂电池充放电循环次数远大于 500 次。
- 多个锂电池可以随意并联使用。

51 单片机开发与应用技术详解

- 无污染，由于锂电池中不含镉、铅、汞等重金属元素，对环境无污染，是理想的绿色电池。
- 快速充电，如果使用额定电压为 4.2V 的恒流恒压充电器，可以使锂离子电池在一至两个小时得到充满。

锂电池与其他可充电电池相比，其价格相对较高。但是随着技术的发展，锂电池的性价比越来越高，目前已广泛应用在各类便携式移动设备上。

38.1.2 锂电池充电概述

锂电池对充电器的要求比较高，为了保护电池和最大化地延长使用寿命，在充电时需要注意如下事项。

- 对锂电池需要进行热保护，防止发热太大而损害锂电池。
- 锂电池充电需要严格控制充电电压和充电电流。
- 为了有效利用电池容量，需将锂电池充电至最大电压。
- 防止过压充电，过压充电对锂电池有损害，严重影响电池寿命。
- 充电结束后应及时关断电源。

为了达到更好的充电效果，一般首先采用预充，然后采用大电流进行快充。当充电达到容量的 90% 后，进行满充，采用小电流涓流充电。在充电过程中，需要采用专业的充电检测芯片来对充电过程进行检测，在充电电路中使用单片机来综合进行管理，可以做到精确的智能控制。使用单片机和充电管理芯片相结合的方法可以有效地保护电池、缩短充电时间并延长电池使用寿命。

38.2 智能充电管理芯片 MAX1898

锂电池智能充电的核心是使用合适的充电管理芯片。目前市场上存在大量的电池充电芯片，它们可直接用于进行充电器的设计。在选择具体的电池充电芯片时，需要注意以下几点。

- 可充电电池的数目，有的充电管理芯片可以对多节锂电池进行充电，有的则只可以对一节锂电池进行充电。
- 充电电压和电流值，充电电流的大小决定了充电的时间，而充电电压不应超过锂电池所规定的充电电压。
- 充电方式：充电过程是快充、慢充还是可控充电过程。

下面介绍一款高性能的锂电池充电管理芯片 MAX1898，其可以对单节锂电池进行充电管理。

38.2.1 智能充电管理芯片 MAX1898 概述

MAX1898 是 MAXIM 公司生产的线性锂电池充电器，可用于手机、PDA 和数码相机等单节锂电池供电的便携式系统中。MAX1898 外部配合一个 PNP 或 PMOS 晶体管，便可以构成一个完整的单节锂电池充电电路。MAX1898 为锂电池提供了精确的恒流和恒压充电，充电电压的精度为 $\pm 0.75\%$ ，这样可以大大提高电池性能并延长电池的使用时间。MAX1898 的充电电流和充电时间可由用户设定，其采用内部检流。MAX1898 为用户提供了全面的充电显示，包括充电状态的输出指示、输入电源是否与充电器连接的输出指示和充电电流指示。除此之外，MAX1898 还具有可选的充电终止安全定时器、输入关断控制、充电周期重启和低电流预充功能。

智能充电管理芯片 MAX1898 的主要特性如下：

- 安全的线性充电方式。
- 使用简单、低成本的 PNP 或 PMOS 作为调整元件。
- 4.5V~12V 的输入电压。
- 内置检流电阻，而无须外部检流。
- 充电电压精度可达 $\pm 0.75\%$ 。
- 自动检测输入电源。
- 具有 LED 充电状态指示。

- 可编程安全定时器。
- 可编程充电电流。
- 检流监视输出。
- 具有自动重启功能。
- 小尺寸 μ MAX 封装，减少电路板面积。

MAX1898 具有很高的集成度，在更小的尺寸内集成了更多的功能，尽可能多地覆盖了基本应用电路，而只需少数外部元件。MAX1898 具有两个版本，可对所有化学类型的 Li+ 电池进行安全充电。电池调节电压为 4.2V (MAX1898EUB42) 或 4.1V (MAX1898EUB41)。两者都采用 10 引脚、超薄型 μ MAX 封装。MAX1898 的引脚封装，如图 38-1 所示。

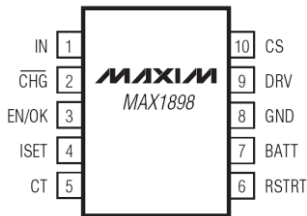


图 38-1 MAX1898 的引脚分配

智能充电管理芯片 MAX1898 的引脚功能如下所示。

- IN (Pin1)：电源输入引脚，检测输入的电压或电流。
- CHG (Pin2)：充电状态指示引脚，可以驱动 LED 来进行状态显示。
- EN/OK (Pin3)：使能输入引脚/输入电源 OK 输出提示引脚。如果作为输入引脚，可以通过 EN 输入禁止芯片工作；如果作为输出引脚，可以通过 OK 提示输入电源是否与充电器连接。
- ISET (Pin4)：充电电流调节引脚。通过连接一个电阻接地来设置充电的最大电流。
- CT (Pin5)：安全充电时间设置引脚。通过连接一个电容接地来设置最大充电时间。当外接电容为 100nF 时，最大充电时间约为三小时。CT 引脚直接接地，将禁用此功能。
- RSTRT (Pin6)：充电自动重新启动控制引脚。RSTRT 引脚直接接地时，如果电池电压下降至基准电压阈值以下 200mV 便自动开始新一轮充电周期。RSTRT 引脚连接一个电阻接地时，可以降低其电压阈值。RSTRT 引脚悬空或 CT 引脚接地时，充电自动重新启动功能被禁用。
- BATT (Pin7)：充电电池输入脚，连接单个锂电池的正极。此引脚需旁路一个大电解电容接地。
- GND (Pin8)：接地引脚。
- DRV (Pin9)：外部 PNP 或 PMOS 晶体管驱动器，接晶体管的基极。
- CS (Pin10)：电流传感输入引脚，接 PNP 或 PMOS 晶体管的发射极。

38.2.2 MAX1898 充电工作原理

智能充电管理芯片 MAX1898 内部集成了输入电压检测器、输入电流调节器、充电电流检测器、定时器、温度传感器和状态控制器。其中，各部分的功能介绍如下：

- 输入电压检测器可以检测输入电源是否与充电器连接。
- 输入电流调节器用于限制电源的总输入电流，包括系统负载电流与充电电流。
- 充电电流监测器用于检测充电电流。
- 定时器用来设定最大充电时间。
- 温度传感器用来监测温度变化，具有温度保护功能。
- 状态控制器用来指示充电状态。

MAX1898 外接充电电源和晶体管，可以对单节锂电池进行安全有效的充电，即使不使用

电感，同样能做到最低的功率损耗。MAX1898 可以实现对过放电电池的预充电，而且具有过压保护和温度保护功能，同时可以设置最大充电时间来为锂电池提供额外的保护。

智能充电管理芯片 MAX1898 的典型充电电路，如图 38-2 所示。

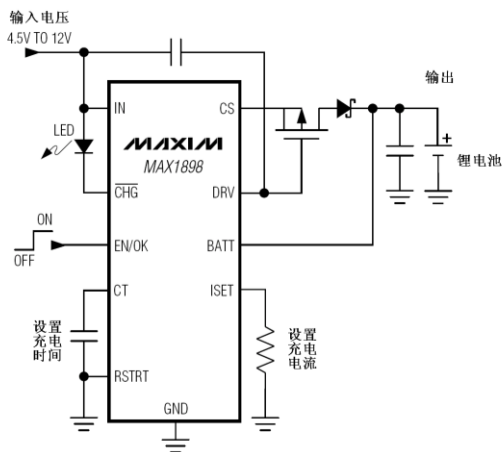


图 38-2 MAX1898 的典型充电电路

其中，输入电源电压范围为 4.5V~12V。为了保证锂电池的恒流恒压充电，输入电源需要采用恒流恒压源。MAX1898 通过外接的场效应晶体管来提供锂电池的充电接口。

MAX1898 的 CT 引脚通过外接的电容 C_{CT} 来设置最大充电时间 t_{CHG} ，也即对锂电池进行快充时的最大充电时间。最大充电时间 t_{CHG} 和定时电容 C_{CT} 的关系如下：

$$t_{CHG} = \frac{C_{CT}}{34.33}$$

其中，最大充电时间 t_{CHG} 的单位为小时，定时电容 C_{CT} 的单位为 nF。

MAX1898 可以在 ISET 引脚外接电阻 R_{SET} 来设置最大充电电流 I_{MAX} ，关系如下：

$$I_{MAX} = \frac{1400}{R_{SET}}$$

其中， R_{SET} 的单位为 Ω ， I_{MAX} 的单位为 A。

MAX1898 的 CHG 引脚连接 LED 指示灯，可以表示不同的充电状态，具体含义如下：

- LED 指示灯灭，表示电池或充电器没有安装。
- LED 指示灯亮，表示预充。
- LED 指示灯亮，表示快充。
- LED 指示灯灭，表示充电结束。
- LED 指示灯以 1.5Hz 频率闪烁，表示充电出错。

当输入电源存在，充电电池在正常的工作温度范围内时，MAX1898 将启动一次充电过程。当充电时间超出片上预置的最大充电时间，或者平均的脉冲充电电流低于设置的快充电流的 20% 时，充电结束。智能充电管理芯片 MAX1898 能够自动检测输入电源，没有输入电源时将自动关断充电电池，以减少电池的漏电。MAX1898 启动快充后，打开外接的 P 沟道场效应管。当 MAX1898 检测到电池电压达到设定的门限时，进入脉冲充电方式，场效应管打开的时间会越来越短，直至充电结束。

38.3 单片机智能控制锂电池充电实例

随着锂电池的普及，常用的手机、数码相机等均采用锂电池及其充电器。下面将通过一个

典型实例，介绍如何使用 51 系列单片机来设计锂电池的智能充电器。该充电器具有预充、充电保护、自动断电和充电状态提示功能。

38.3.1 电路图

这里采用 AT89S52 单片机来控制 MAX1898, 从而实现单节锂电池的智能充电。系统电路图如图 38-3 所示。该电路中所使用的元器件清单, 如表 38-1 所示。

表 38-1 元器件列表

元 器 件	数 值	数 量	元 器 件	数 值	数 量
U1	AT89S52	1 个	R2	200 Ω	1 个
U3	SN74LS04D	1 个	R3	10k Ω 排阻	1 个
U4	HCPL-2601	1 个	R4	33 Ω	1 个
U5	MAX1898	1 个	R5	500 Ω	1 个
C1、C2	30pf	2 个	R6	2.8k Ω	1 个
C3、C12	10 μ f 电解	1 个	Y1	11.0592MHz	1 个
C8	15pF	1 个	LS1	蜂鸣器	1 个
C9	0.1 μ f	1 个	D1	绿色发光二极管	1 个
C10	220 μ f	1 个	D2	红色发光二极管	1 个
C11	100nf	1 个	Q1	PNP 晶体管	1 个
S1	按键开关	1 个	D3	二极管	1 个
R1	5k Ω	1 个	BT1	锂电池	1 个

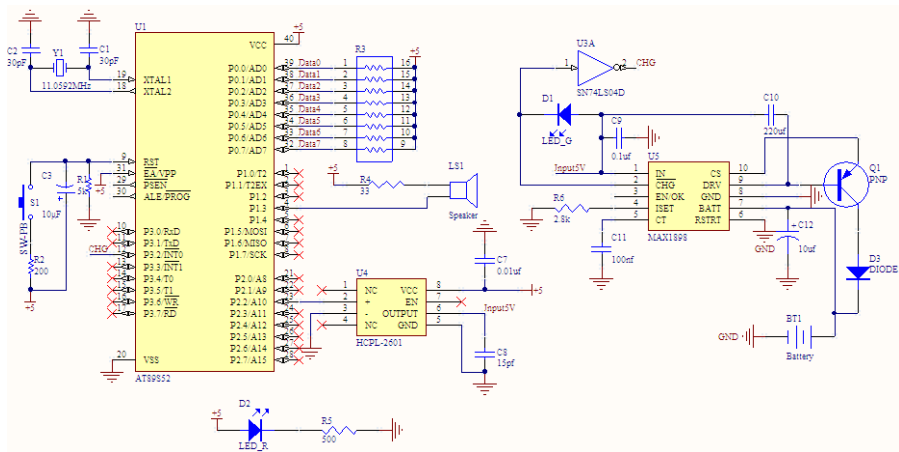


图 38-3 电路原理图

其中，MAX1898 用于对锂电池 BT1 进行充电，电池可以选择一般的手机或数码相机的锂电池。单片机用来实现充电器的智能化控制，比如自动断电、充电完成报警提示等。蜂鸣器 LS1 连接单片机的 P1.3 引脚，用来输出报警信号。光耦模块 HCPL-2601 用来在充电结束时自动断电。芯片 HCPL-2601 的引脚分配，如图 38-4 所示。

光耦模块 HCPL-2601 的引脚功能如下:

- NC (Pin1、Pin4): 悬空。
- + (Pin2): 发光二极管的正极。
- - (Pin3): 发光二极管的负极。
- GND (Pin5): 接地引脚。

51 单片机开发与应用技术详解

- V_O (Pin6): 输出引脚。
- V_E (Pin7): 使能引脚。 V_E 为低电平时, 无论有无输入, V_O 输出都为高。不使用该引脚时, 悬空即可。

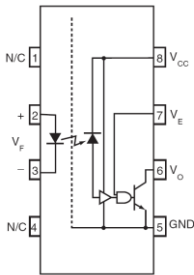


图 38-4 HCPL-2601 的引脚分配

- V_{CC} (Pin8): 电源输入引脚。

单片机的 P2.2 引脚控制光耦模块 HCPL-2601, 在充电结束时可以及时关断充电电源。MAX1898 的 \overline{CHG} 引脚通过反相器 SN74LS04D 连接至单片机的外部中断 0 引脚 P3.4。发光二极管 D1 连接 \overline{CHG} 引脚, 用来显示充电状态。

MAX1898 的充电电流和最大充电时间按照如下的参数设置。

- R6 设置充电电流的电阻, 阻值为 $2.8k\Omega$, 即设置最大充电电流为 500mA。
- C11 设置充电时间的电容, 容值为 100nf, 设置最大充电时间为三小时。

38.3.2 智能充电器的功能

这里所设计的智能充电器, 需要完成预充、快充、满充、断电和报警等功能。这些功能主要依靠智能充电管理芯片 MAX1898 和单片机 AT89S52 来共同实现, 下面分别进行介绍。

➤ 预充

首先, 系统上电后, 当 MAX1898 检测到外接的锂电池时, 将定时器复位, 从而进入预充状态。在进行预充时, MAX1898 以快充电流的 10% 给锂电池充电, 使电源电压、温度恢复到正常状态。

预充时间由 MAX1898 外接电容 C11 决定, 这里选择 100nF, 即预充时间为 45 分钟。如果在预充时间内锂电池电压达到 2.5V 以上, 且锂电池温度正常, 则 MAX1898 进入快充过程; 如果在预充时间内锂电池电压仍低于 2.5V, 则认为该锂电池不可充电, \overline{CHG} 引脚连接的发光二极管闪烁, 表示充电故障。

➤ 快充

当预充结束后, 锂电池电压上升到 2.5V 以上, MAX1898 便以恒定电路对锂电池进行快充。快充也称为恒流充电。在快充过程中, 锂电池电压逐渐上升, 直至锂电池电压达到所设定的终止电压。

➤ 满充

当在快充过程中, 锂电池电压达到所设定的终止电压, 快充结束, 充电电流快速递减, 充电进入满充过程。此时, 当充电速率降到设置值以下或满充时间超时, 进入顶端截止充电过程。

在顶端截止充电时, MAX1898 以极小的充电电流为电池补充能量。一般情况下, 满充和顶端截止充电过程可以延长锂电池 5%~10% 的使用时间。

➤ 断电

当锂电池电量充满后, MAX1898 芯片的 \overline{CHG} 引脚输出低电平, 通过反相器触发单片机的中断。此时, 单片机如果判断充电完毕, 则通过 P2.2 引脚控制光耦模块 HCPL-2601, MAX1898 的输入电源。从而保证锂电池不会被过充电, 同时也可以减小系统功耗。

➤ 报警

当单片机切断 MAX1898 的输入电源后, 通过 P1.3 引脚触发蜂鸣器报警, 从而提示用户电池已充满, 应及时取出电池。至此, 整个充电过程便结束。

38.3.3 建立项目

首先在 Keil μ Vision3 集成开发环境中建立项目, 具体操作步骤如下:

- (1) 首先打开 μ Vision3，在 μ Vision3 中，选择“Project”→“New”→“ μ Vision Project”命令，新建一个工程，并保存。
- (2) 在弹出的选择器件对话框中选择 Atmel 公司的 AT89S52，如图 38-5 所示。
- (3) 单击“确定”按钮，此时弹出“ μ Vision3”对话框，如图 38-6 所示。单击“是”按钮，完成工程的建立。
- (4) 选择“File”→“New”命令，新建一个程序文件，并保存为*.C 文件，可以在其中输入程序代码。

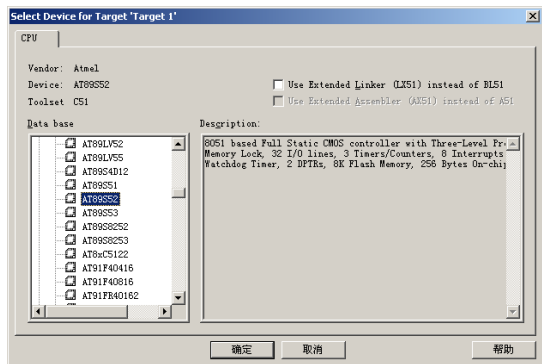


图 38-5 选择单片机 AT89S52



图 38-6 “ μ Vision3”对话框

38.3.4 程序设计

这里通过单片机 AT89S52 来控制 MAX1898，从而实现智能充电管理。主函数中，首先初始化定时器/计数器和外部中断。外部中断 0 由 MAX1898 的 CHG 引脚触发，在其服务程序中启动定时器。定时器/计数器 T0 的中断例程，用来判断充电是否结束。当充电结束时，切断 MAX1898 的输入电源，并触发蜂鸣器报警。程序代码示例如下：

```
#include <reg52.h> //头文件
#include <stdio.h>

#define uint unsigned int

sbit HCPL = P2^2; //光耦模块 HCPL-2601 输出控制
sbit BP = P1^3; //蜂鸣器报警控制

uint CountT0; //定时器 T0 中断次数
uint CountINT0; //外部中断 0 次数

void timer0() interrupt 1 //定时器/计数器 T0 中断服务例程
{
    TR0 = 0; //停止计数

    TH0 = 19; //重置初值
    TL0 = 136;

    CountT0++; //定时器 T0 中断次数加 1

    if (CountT0>620)
    {
        if (CountINT0==1) //如果外部中断 0 次数为 1，则认为充电完毕
        {
            HCPL = 0; //关闭 MAX1898 输入电源
            BP = 0; //触发蜂鸣器报警
            printf("Bettery is charged over.");
        }
    }
}
```

```

else //否则，表示充电过程出错
{
    HCPL = 1; //保持 MAX1898 输入电源
    BP = 1; //蜂鸣器不报警
    printf("There is something error.");
}

ET0 = 0; //关闭定时器 T0 中断
EX0 = 0; //关闭外部中断 0

CountINT0 = 0; //计数器清 0
CountT0 = 0; //计数器清 0
}
else
{
    TR0 = 1; //启动定时器 T0 计数
}
}

void int0() interrupt 0 //外部中断 0 服务例程
{
    if (CountINT0==0) //定时器 T0 未中断
    {
        TH0 = 19; //设置定时初值
        TL0 = 136;
        CountT0 = 0; //计数器清 0
        TR0 = 1; //启动定时器/计数器 0
    }
    CountINT0++; //计数器清 0
}

void main()
{
    TMOD = 0x01; //定时器模式 1，T0 为 16 位定时/计数器
    PT0 = 1; //设置 T0 中断为高优先级
    IT0 = 1; //设置外部中断 0 为边沿触发

    ET0 = 1; //打开 T0 中断
    EX0 = 1; //打开外部中断 0
    EA = 1; //打开 CPU 中断

    HCPL = 1; //MAX1898 输入电源，开始充电
    BP = 1; //关闭蜂鸣器

    CountINT0 = 0; //计数器清 0
    CountT0 = 0; //计数器清 0

    printf("Begin battery charge.");
    while(1) //主循环，充电过程
    {
    }
}
```

当程序设计完毕后，便可以在实际硬件上工作，来对锂电池进行充电。充电过程中，通过发光二极管 D1 及蜂鸣器的状态，可以判断充电是否结束。

38.4 小结

本章首先介绍了广泛使用的锂电池，以及锂电池的充电要求。接着介绍了 MAXIM 公司的一款高性能的智能充电管理芯片 MAX1898，包括 MAX1898 引脚功能及其充电工作原理。最

后，本章通过一个具体的实例，介绍了如何使用 51 系列单片机控制 MAX1898 来实现单节锂电池的智能充电过程。锂电池及其充电器广泛应用于生活中，因此，熟练掌握本章内容具有极大的实际意义。

《51 单片机开发与应用技术详解》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1. 短信

您只需编写如下短信：B07920+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2. 电子邮件

您可以发邮件至jsj@phei.com.cn**或**editor@broadview.com.cn**。**

3. 信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview[®]
www.broadview.com.cn

(1) 您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

(2) 您了解新书信息的途径、影响您购买图书的因素；

(3) 您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需发送“B07920+退订”至10666666789即可，

或者编写邮件“B07920+退订+手机号码+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn 亦可取

消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站(www.broadview.com.cn)上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路173信箱 博文视点(100036) 电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396; (010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036